# VMM User Guide

July 2011

# Copyright Notice and Proprietary Information

Copyright © 2011 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

## Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

> "This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of _____ and its employees. This is copy number _____."

## Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

## Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

## Registered Trademarks (®)

Synopsys, AEON, AMPS, Astro, Behavior Extracting Synthesis Technology, Cadabra, CATS, Certify, CHIPit, CoMET, Confirma, CODE V, Design Compiler, DesignWare, EMBED-IT!, Formality, Galaxy Custom Designer, Global Synthesis, HAPS, HapsTrak, HDL Analyst, HSIM, HSPICE, Identify, Leda, LightTools, MAST, METeor, ModelTools, NanoSim, NOVeA, OpenVera, ORA, PathMill, Physical Compiler, PrimeTime, SCOPE, Simply Better Results, SiVL, SNUG, SolvNet, Sonic Focus, STAR Memory System, Syndicated, Synplicity, the Synplicity logo, Synplify, Synplify Pro, Synthesis Constraints Optimization Environment, TetraMAX, UMRBus, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

## Trademarks (™)

AFGen, Apollo, ARC, ASAP, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, BEST, Columbia, Columbia-CE, Cosmos, CosmosLE, CosmosScope, CRITIC, CustomExplorer, CustomSim, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, DesignPower, DFTMAX, Direct Silicon Access, Discovery, Eclypse, Encore, EPIC, Galaxy, HANEX, HDL Compiler, Hercules, Hierarchical Optimization Technology, High-performance ASIC Prototyping System, HSIM$^{plus}$, i-Virtual Stepper, IICE, in-Sync, iN-Tandem, Intelli, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Macro-PLUS, Magellan, Mars, Mars-Rail, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, MultiPoint, ORAengineering, Physical Analyst, Planet, Planet-PL, Polaris, Power Compiler, Raphael, RippledMixer, Saturn, Scirocco, Scirocco-i, SiWare, Star-RCXT, Star-SimXT, StarRC, System Compiler, System Designer, Taurus, TotalRecall, TSUPREM-4, VCSi, VHDL Compiler, VMC, and Worksheet Buffer are trademarks of Synopsys, Inc.

## Service Marks ($^{sm}$)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.
ARM and AMBA are registered trademarks of ARM Limited.
Saber is a registered trademark of SabreMark Limited Partnership and is used under license.
All other product or company names may be trademarks of their respective owners.

# Contents


## 1 Introduction

Overview . . . . . 1-2
- VMM Benefits: . . . . . 1-4
- Ease of Use . . . . . 1-5
- Reuse. . . . . 1-6
- Effectiveness . . . . . 1-7

How to Use This User Guide? . . . . . 1-8
Basic Concepts of VMM . . . . . 1-9
- Building Blocks - Class Library . . . . . 1-10
- Verification Environments and Execution Control Phases. . . . . 1-12
- Enhanced Verification Performance and Flexibility . . . . . 1-14
- Debug and Analysis: Message Service Class and Transaction Debug . 1-15

What's New in VMM? . . . . . 1-16
UML Diagram . . . . . 1-17
Resources . . . . . 1-18

## 2 Architecting Verification Environments

Overview . . . . . 2-2
Testbench Architecture . . . . . 2-2
- Signal Layer . . . . . 2-6
- Command Layer . . . . . 2-15

Functional Layer . . . . . 2-17
Scenario Layer . . . . . 2-20
Test Layer . . . . . 2-22
Sub-environments . . . . . 2-23
Constructing and Controlling Environments . . . . . 2-28
Quick Transaction Modeling Style . . . . . 2-30
Understanding Implicit and Explicit Phasing . . . . . 2-31
Composing Explicitly Phased Environments . . . . . 2-33
Composing Explicitly Phased Sub-Environments . . . . . 2-41
Composing Implicitly Phased Environments/Sub-Environments . . . . . 2-48
Reaching Consensus for Terminating Simulation . . . . . 2-56
Architecting Verification IP (VIP) . . . . . 2-63
VIP and Testbench Components . . . . . 2-63
Transactions . . . . . 2-63
Transactors . . . . . 2-65
Communication . . . . . 2-71
Environments and Sub-Environments . . . . . 2-73
Testing VIPs . . . . . 2-73
Advanced Usage . . . . . 2-75
Mixed Phasing . . . . . 2-75

# 3 Modeling Transactions

Overview . . . . . 3-2
Class Properties/Data Members . . . . . 3-5
Quick Transaction Modeling Style . . . . . 3-5
Message Service in Transaction . . . . . 3-6
Randomizing Transaction Members . . . . . 3-7
Context References . . . . . 3-8
Inheritance and OOP . . . . . 3-10
Handling Transaction Payloads . . . . . 3-15
Methods . . . . . 3-17
Factory Service for Transactions . . . . . 3-20

Constraints . . . . . 3-20
Shorthand Macros . . . . . 3-24
User-Defined Implementations . . . . . 3-25
Unsupported Data Types . . . . . 3-31
rand_mode() copy in Shorthand Macros . . . . . 3-34

# 4 Modeling Transactors and Timelines

Overview . . . . . 4-2
Transactor Phasing . . . . . 4-6
Explicit Transactor Phasing . . . . . 4-7
Implicit Phasing . . . . . 4-14
Threads and Processes Versus Phases . . . . . 4-18
Physical-Level Interfaces . . . . . 4-22
Transactor Callbacks . . . . . 4-26
Advanced Usage . . . . . 4-31
User-defined vmm_xactor Member Default Implementation . . . . . 4-31
User-Defined Implicit Phases . . . . . 4-32
Skipping an Implicit Phase . . . . . 4-35
Disabling an Implicit Component . . . . . 4-35
Synchronizing on Implicit Phase Execution . . . . . 4-36
Breakpoints on Implicit Phasing . . . . . 4-38
Concatenation of Tests . . . . . 4-40
Explicitly Phasing Timelines . . . . . 4-42

# 5 Communication

Overview . . . . . 5-2
Channel . . . . . 5-2
Channel Declaration (vmm_channel_typed) . . . . . 5-4
Channel Declaration (vmm_channel) . . . . . 5-5
Connection of Channels Between Transactors . . . . . 5-5
Declaring Factory Enabled Channels . . . . . 5-7

Overriding Channel Factory . . . . . 5-8
Channel Completion and Response Models . . . . . 5-9
Typical Channel Execution Model . . . . . 5-9
Channel Record/Playback . . . . . 5-13
Completion Using Notification (vmm_notify) . . . . . 5-15
Notification Service Class . . . . . 5-16
Notify Observer . . . . . 5-18
Transport Interfaces in OSCI TLM2.0 . . . . . 5-19
Blocking Transport . . . . . 5-20
Non-Blocking Transport . . . . . 5-22
Sockets . . . . . 5-24
Connecting Blocking Components to Non-blocking Components . . . . . 5-27
Generic Payload . . . . . 5-29
Broadcasting Using TLM2.0 . . . . . 5-31
Analysis Port Usage with Many Observers . . . . . 5-32
Analysis Port Multiple Ports Per Observer . . . . . 5-34
Shorthand Macro IDs . . . . . 5-34
Peer IDs . . . . . 5-36
Interoperability Between vmm_channel and TLM2.0 . . . . . 5-37
Connecting vmm_channel and TLM interface . . . . . 5-38
TLM2.0 Accessing Generators . . . . . 5-39
Forward Path Non-Blocking Connection . . . . . 5-40
Bidirectional Non-Blocking Connection . . . . . 5-42
Advanced Usage . . . . . 5-43
Updating Data in Analysis Ports From vmm_notify . . . . . 5-43
Connect Utility (vmm_connect) . . . . . 5-45
Channel Non-Atomic Transaction Execution . . . . . 5-47
Channel Out-of-Order Atomic Execution Model . . . . . 5-49
Channel Passive Response . . . . . 5-54
Channel Reactive Response . . . . . 5-57
vmm_tlm_reactive_if . . . . . 5-61

# 6 Implementing Tests & Scenarios


Overview . . . . . 6-2
Generating Stimulus . . . . . 6-2
- Random Stimulus. . . . . 6-4
- Directed Stimulus. . . . . 6-10
- Generating Exceptions. . . . . 6-13
- Embedded Stimulus. . . . . 6-18
- Controlling Random Generation . . . . . 6-20

Modeling Scenarios. . . . . 6-24
- Architecture of the Generators. . . . . 6-25
- Scenario Selection. . . . . 6-26

Modeling Generators. . . . . 6-28
- Atomic Generation . . . . . 6-28
- Multiple-Stream Scenarios. . . . . 6-29
- Single-Stream Scenarios . . . . . 6-45
- Parameterized Atomic and Scenario Generators . . . . . 6-52

Implementing Testcases . . . . . 6-54
- Creating an Explicitly Phased Test . . . . . 6-55
- Creating an Implicitly Phased Test . . . . . 6-55
- Running Tests . . . . . 6-56


# 7 Common Infrastructure and Services


Common Object. . . . . 7-2
- Overview . . . . . 7-2
- Setting Object Relationships . . . . . 7-2
- Finding Objects . . . . . 7-6
- Printing and Displaying Objects. . . . . 7-7
- Object Traversing. . . . . 7-8
- Namespaces . . . . . 7-9

Message Service. . . . . 7-10
- Overview . . . . . 7-11

Message Source . . . . . 7-12
Message Type . . . . . 7-12
Message Severity . . . . . 7-13
Message Filters . . . . . 7-14
Simulation Handling . . . . . 7-14
Shorthand Macros . . . . . 7-15
Issuing Messages . . . . . 7-16
Filtering Messages . . . . . 7-17
Redirecting Message to File . . . . . 7-19
Promotion and Demotion . . . . . 7-20
Message Catcher . . . . . 7-20
Message Callbacks . . . . . 7-23
Stop Simulation Depending Upon Error Number . . . . . 7-25
Class Factory Service . . . . . 7-25
Overview . . . . . 7-26
Modeling a Transaction to be Factory Enabled . . . . . 7-28
Creating Factories . . . . . 7-31
Replacing Factories . . . . . 7-32
Factory for Parameterized Classes . . . . . 7-34
Factory for Atomic Generators . . . . . 7-36
Factory for Scenario Generators . . . . . 7-38
Modifying a Testbench Structure Using a Factory . . . . . 7-41
Options & Configurations Service . . . . . 7-42
Overview . . . . . 7-42
Hierarchical Options (vmm_opts) . . . . . 7-43
Specifying Placeholders for Hierarchical Options . . . . . 7-44
Setting Hierarchical Options . . . . . 7-44
Setting Hierarchical Options on Command Line . . . . . 7-45
Structural Configurations . . . . . 7-47
Specifying Structural Configuration Parameters in Transactors . . . . . 7-49
Setting Structural Configuration Parameters . . . . . 7-50
Setting Options on Command Line . . . . . 7-51

RTL Configuration . . . . . 7-52
Defining RTL Configuration Parameters . . . . . 7-53
Using RTL Configuration in vmm_unit Extension . . . . . 7-55
First Pass: Generation of RTL Configuration Files . . . . . 7-56
Second Pass: Simulation Using RTL Configuration File . . . . . 7-57
Simple Match Patterns . . . . . 7-57
Overview . . . . . 7-57
Pattern Matching Rules . . . . . 7-58

# 8 Methodology Guide

Recommendations. . . . . 8-1
Transactions . . . . . 8-1
Message Service . . . . . 8-2
Transactors . . . . . 8-3
Callbacks . . . . . 8-3
Channels . . . . . 8-4
Environments . . . . . 8-4
Tests and Generators . . . . . 8-5
Channels and TLM Ports . . . . . 8-6
Configuration . . . . . 8-6
Rules . . . . . 8-7
Transactions . . . . . 8-7
Message Service . . . . . 8-8
Transactors . . . . . 8-8
Callbacks . . . . . 8-10
Channels . . . . . 8-10
Environments . . . . . 8-12
Notifications . . . . . 8-14
Tests and Generators . . . . . 8-15

# 9 Optimizing, Debugging and Customizing VMM


Optimizing VMM Components . . . . . 9-2
Garbage-Collecting vmm_object Instances . . . . . 9-2
Optimizing vmm_log Usage . . . . . 9-3
Static vmm_log Instances . . . . . 9-4
vmm_log Instances in vmm_channel . . . . . 9-6
Customizing VMM . . . . . 9-6
Adding to the Standard Library . . . . . 9-7
Customizing Base Classes . . . . . 9-8
Symbolic Base Class . . . . . 9-9
Customizing Utility Classes . . . . . 9-12
Symbolic Utility Class . . . . . 9-13
Underpinning Classes . . . . . 9-14
Base Classes as IP . . . . . 9-17


# 10 Primers


Multi-Stream Scenario Generator Primer . . . . . 10-2
Introduction . . . . . 10-2
Step1: Creation of Scenario Class . . . . . 10-3
Step 2: Usage of Logical Channels in MSS . . . . . 10-4
Step 3: Registration of MSS in MSSG . . . . . 10-5
Complete Example of a Simple MSSG . . . . . 10-6
Class Factory Service Primer . . . . . 10-10
Introduction . . . . . 10-10
Step 1: Modeling Classes to be Factory Ready . . . . . 10-11
Step 2: Instantiating a Factory in Transactor . . . . . 10-14
Step 3: Instantiating a MSS Factory in MSSG . . . . . 10-15
Step 4: Replacing a Factory . . . . . 10-16
Step 4a: Replacing a Factory by a New One . . . . . 10-17
Step 4b: Replacing a Factory by a Copy . . . . . 10-19
Summary . . . . . 10-20

Hierarchical Configuration Primer . . . . . 10-22
Introduction . . . . . 10-22
Step 1: Setting/Getting Global Options . . . . . 10-24
Step 2: Setting/Getting Hierarchical Options . . . . . 10-25
Step 3: Getting Structural Options . . . . . 10-26
Step 4: Setting Options . . . . . 10-29
Step 4a: Setting Options with `set_*` . . . . . 10-29
Step 4b: Setting Options in Command Line . . . . . 10-30
Step 4c: Setting Options With Command File . . . . . 10-30
Conclusion . . . . . 10-31
RTL Configuration Primer . . . . . 10-32
Introduction . . . . . 10-32
Step 1: Defining RTL Configurations . . . . . 10-34
Step 2: Nested RTL Configurations . . . . . 10-35
Step 3: Instantiating RTL Configurations . . . . . 10-35
Step 4: Generating RTL Configuration File . . . . . 10-37
Step 5: Simulation Using RTL Configuration File . . . . . 10-38
Conclusion . . . . . 10-39
Implicitly Phased Master Transactor Primer . . . . . 10-40
Introduction . . . . . 10-40
The Protocol . . . . . 10-40
The Verification Components . . . . . 10-41
Step 2: Instantiating and Connecting the DUT . . . . . 10-44
Step 3: Modeling the APB Transaction . . . . . 10-45
Step 4: Modeling the Master Transactor . . . . . 10-47
Step 5: Implementing an Observer . . . . . 10-56
Step 6: Instantiating the Components in the Environment . . . . . 10-56
Step 7: Implementing Sanity Test . . . . . 10-58
Step 8: Adding Debug Messages . . . . . 10-60
Step 9: Implementing Transaction Generator . . . . . 10-61
Step 10: Implementing the Top-Level File . . . . . 10-61

# A Standard Library Classes (Part 1)

VMM Standard Library Class List . . . . . A-2
factory . . . . . A-4
vmm_atomic_gen#(T) . . . . . A-13
<class-name>_atomic_gen_callbacks . . . . . A-26
vmm_atomic_scenario#(T) . . . . . A-29
vmm_broadcast . . . . . A-30
vmm_channel . . . . . A-44
    VMM Channel Relationships . . . . . A-45
    VMM Channel Record or Replay . . . . . A-47
vmm_channel_typed#(type) . . . . . A-102
vmm_connect#(T,N,D) . . . . . A-110
vmm_consensus . . . . . A-117
vmm_data . . . . . A-147
vmm_env . . . . . A-207
vmm_group . . . . . A-245
vmm_group_callbacks . . . . . A-247
vmm_log . . . . . A-254
vmm_log_msg . . . . . A-310
vmm_log_callback . . . . . A-320
vmm_log_catcher . . . . . A-326
vmm_log_format . . . . . A-333
vmm_ms_scenario . . . . . A-341
vmm_ms_scenario_gen . . . . . A-351
vmm_notification . . . . . A-418
vmm_notify . . . . . A-423
vmm_notify_callbacks . . . . . A-448
vmm_notify_observer#(T,D) . . . . . A-451
vmm_object . . . . . A-455
vmm_object_iter . . . . . A-490
vmm_opts . . . . . A-494

# B Standard Library Classes (Part 2)

VMM Standard Library Class List . . . . . B-2
vmm_phase . . . . . B-5
vmm_phase_def . . . . . B-18
vmm_rtl_config_DW_format . . . . . B-37
vmm_rtl_config . . . . . B-38
vmm_rtl_config_file_format . . . . . B-49
vmm_scenario . . . . . B-61
vmm_scenario_gen#(T, text) . . . . . B-95
<class-name>_scenario . . . . . B-139
<class-name>_atomic_scenario . . . . . B-158
<class-name>_scenario_election . . . . . B-161
<class-name>_scenario_gen_callbacks . . . . . B-170
vmm_scheduler . . . . . B-175
vmm_scheduler_election . . . . . B-191
vmm_ss_scenario#(T) . . . . . B-204
vmm_simulation . . . . . B-205
vmm_subenv . . . . . B-214
vmm_test . . . . . B-246
vmm_test_registry . . . . . B-257
vmm_timeline . . . . . B-261
vmm_timeline_callbacks . . . . . B-285
vmm_tlm . . . . . B-288
vmm_tlm_extension_base . . . . . B-290
vmm_tlm_generic_payload . . . . . B-291
vmm_tlm_analysis_port#(I,D) . . . . . B-300
vmm_tlm_analysis_export#(T,D) . . . . . B-302
‘vmm_tlm_analysis_export(SUFFIX) . . . . . B-304
vmm_tlm_b_transport_export#(T,D) . . . . . B-305
vmm_tlm_b_transport_port #(I,D) . . . . . B-310
vmm_tlm_export_base #(D,P) . . . . . B-313
vmm_tlm_nb_transport_bw_export#(T,D,P) . . . . . B-325
vmm_tlm_nb_transport_bw_port#(I,D,P) . . . . . B-330
vmm_tlm_nb_transport_export#(T,D,P) . . . . . B-333
vmm_tlm_nb_transport_fw_export#(T,D,P) . . . . . B-336

vmm_tlm_nb_transport_fw_port#(I,D,P) . . . . . B-341
vmm_tlm_nb_transport_port#(I,D,P) . . . . . B-344
vmm_tlm_port_base#(D,P) . . . . . B-347
vmm_tlm_initiator_socket#(I,D,P) . . . . . B-356
vmm_tlm_target_socket#(T,D,P) . . . . . B-359
vmm_tlm_transport_interconnect#(DATA) . . . . . B-363
vmm_tlm_transport_interconnect_base#(DATA,PHASE) . . . . . B-365
vmm_tlm_reactive_if #(DATA, q_size) . . . . . B-370
vmm_unit . . . . . B-377
vmm_version . . . . . B-406
vmm_voter . . . . . B-413
vmm_xactor . . . . . B-417
Summary . . . . . B-417
vmm_xactor_callbacks . . . . . B-477
vmm_xactor_iter . . . . . B-478
Using the vmm_xactor_iter Class . . . . . B-479
Using the Shorthand Macro `foreach_vmm_xactor() . . . . . B-480

C **Command Line Reference**

D **Release Notes**

New Features in VMM User Guide . . . . . D-1
New Base Classes . . . . . D-1

# 1

# Introduction

The Verification Methodology Manual (VMM) describes the framework for developing re-usable verification components, sub-environments and environments.

This framework enables higher productivity, reuse and interoperability. VMM provides a class library and defines industry best practices with coding guidelines and rules. The set of guidelines and recommendations paves the path for creating highly efficient transaction-level, constrained-random verification environments using SystemVerilog.

This chapter introduces the main concepts of VMM and its usage models in the following sections:


- “Overview”
- “How to Use This User Guide?”

- “Basic Concepts of VMM”
- “What's New in VMM?”
- “UML Diagram”
- “Resources”


## Overview

Winning in competitive electronic systems and computer industries requires continuous delivery of high quality and feature-rich products efficiently. To this end, companies constantly seek innovative ways to improve their product development cycles.

Electronic designs have become so complex that design development often relies on ready-made foundations of design and verification blocks. This translates into the requirement of even more complex verification components and environments. With an ever-shrinking time-to-market window, verification task has become crucial within the complex system and chip design flow.

Companies strive to raise productivity and quality of design verification, streamline and reduce the time it takes to functionally validate a design before fabrication.

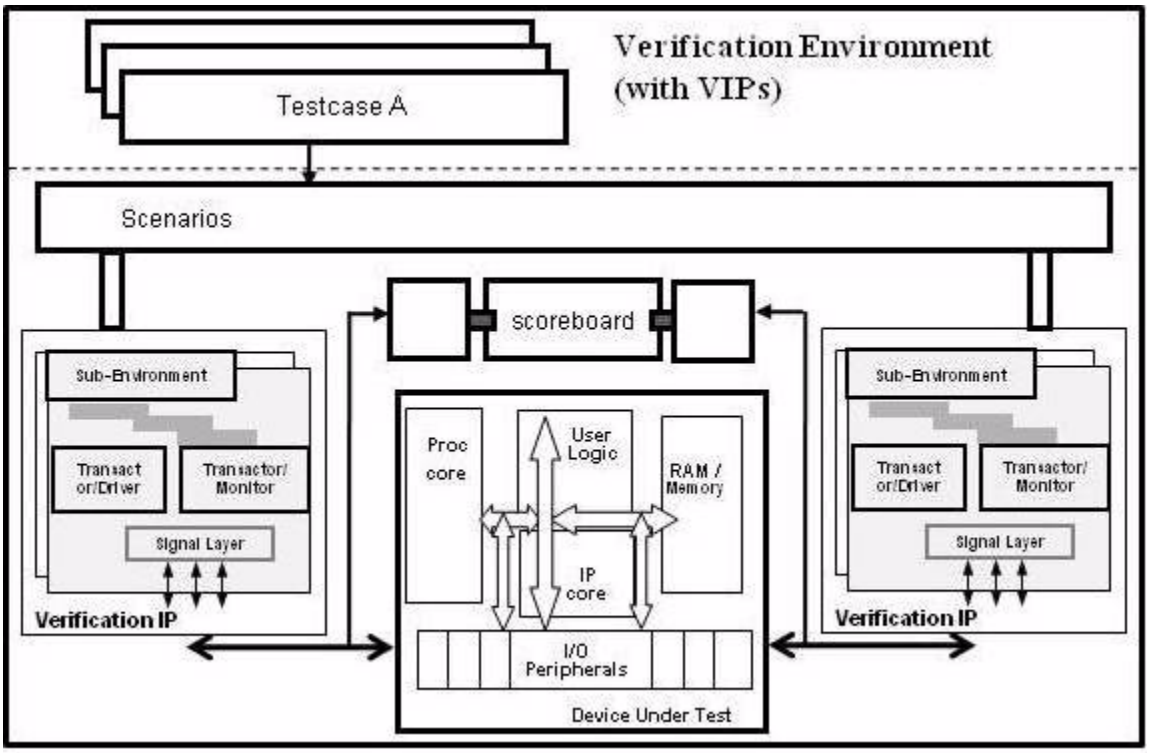
We see that today's chip designs require work at many levels of abstractions - high-level abstract models, transaction-level models and gate level netlist.

Design components in many levels of abstractions are frequently reused and expanded. Complications in their integration - be it internal design blocks or third party IPs, together with their verification environments, can unexpectedly delay the development.

A well structured verification environment and its components such as verification IPs, should smoothen the path for integration, capability for horizontal and vertical reuse. It should also offer flexibility to create tests for verifying various design configurations, all design operating modes and to generate meaningful information for debugging.

Figure 1-1 is an example for using VIPs for verifying design under test (DUT).

*Figure 1-1 Verification Environment Using Verification IPs With DUT.*




A layered verification architecture such as VMM, uses the following flow to provide flexibility and reusability for development of testbenches for use from block level to chip and system-level verification.

Based on a robust verification plan, the test components define specific configurations and requirements. The test information provides the means for transactors to create transactions used by a generator to produce random sequences of transactions. A monitor gathers information from transactions passed through the Design Under Test (DUT). A scoreboard is used to compare observed transactions against expected results. Functional coverage measures the verification requirements that have been actually met by the tests.

When verification engineers build a verification environment on top of a well defined and structured base, the overall development is faster. The verification task can quickly shift to generation of tests and scenarios that stimulate the DUT for unearthing hidden bugs. This is possible only if the actual implementation of a verification environment follows well defined standard guidelines.

---

## VMM Benefits:

By using VMM, you can take advantage of the following benefits:

- Avoid common implementation mistakes
- Set clear expectations for verification components and features
- Reduce development time, integration time, and engineers' ramp up time as a result of the known expectations

VMM guidelines help you to develop a well defined and thought-through verification environment that is,

- Easy to use (modular, flexible, customizable)
- Reusable (from block level to top level; from one project to another project)

- Effective (help identify design bugs faster, optimized for superior performances, easy to debug)

## Ease of Use

- **Modularity:** The layered development of a verification environment lets you create logically partitioned components which can be connected with minimum effort. Each verification component serves a specific purpose and performs a specific set of functions. Components such as verification IPs, form the building blocks of a verification environment.
- **Flexibility:** The test stimulus generation using built-in classes and generators provide complete flexibility for tests that cover the entire functionality of the DUT. The ability to mix implicit and explicit phasing promotes complete flexibility and reuse of verification components. The class factory service provides faster stimulus configuration and reuse. Configuration options provide flexibility to control testbench functionality from the runtime command line.
- **Customization:** The ability to weave a user-defined class library into the standard library allows you to provide highly specialized specific features and capabilities that might be missing in the standard version.

## Reuse

- **Horizontal**: You can reuse the environment components between projects. This is made possible by the underlying methodology and layered-base architecture, which enables reuse of transactors, verification components and IPs. The compliance tests and standard protocol scenarios can be reused across projects and design implementations.
- **Vertical**: You can reuse the environment components from block-level to subsystem-level and system-level verification. This is made possible by the sub-environment architecture, which enables easy vertical reuse. Transactor phases can be automatically run or called in the environment. This can be implicitly or explicitly controlled respectively. Both explicitly and implicitly phased sub-environments provide this vertical reuse functionality. Implicitly phased environments simplify incorporation of user-defined phases, addition, deletion and reordering of phases in transactors. Multiple timelines, reuse of verification environments and components achieve fine-grained controllability over phasing in a sub-system.
- **Diagonal**: You can reuse the environment components by various platforms such as, RTL simulation, hardware acceleration and virtual prototyping. The Register Abstraction Layer (RAL) and Hardware Abstraction Layer (HAL) packages provide mechanisms for leveraging testcases and sequences for diagonal reuse. Based on the VMM methodology, these utilities can enhance and ease software debug. RAL provides a unified register-modeling scheme that can be fully reused in various verification environments. HAL provides reuse of existing verification environments for hardware and virtual platforms. For details, see the VMM Application Library user guide.

## Effectiveness

- **Bug-finding methodology**: Increase in design complexity has made constrained-random test generation and functional coverage analysis an essential part of a verification environment. Constrained random stimulus enables the automation of creating a huge number of test scenarios, which would be impossible to replicate manually. VMM provides sound guidelines for efficient modeling of transactors and constraining transactions. Given tests results, functional coverage analysis provides an indication for test quality and verification goals completion.
- **Optimized for performance**: VMM classes have been architected for peak performance, avoiding run-time interpretations and expression evaluation. Additionally, for further improvements in compilation and simulation performance, you can easily turn off some features that are not used.
- **Debug**: Because intricacy and complexity of testcases and scenarios immensely stress the DUT, it is crucial to leverage from tools and mechanisms for debugging the environment and stimulus at higher level of abstraction. VMM provides consistent use of message logging, recording and viewing of transactions and components states. These facilities help debug of complex designs and tests.

## How to Use This User Guide?

The following sections provide a practical usage overview of the VMM core functionality:

- Chapter 2, "Architecting Verification Environments", introduces best practices and usage of base classes to create layered verification environment and components. An overview of creating sub-environments, controlling transactors within an environment is reviewed here.
- Chapter 3, "Modeling Transactions", describes guidelines for modeling transactions.
- Chapter 4, "Modeling Transactors and Timelines", reviews the basic transactor modeling techniques with explicit/implicit phasing. Callback mechanisms and shorthand macros usage are reviewed here.
- Chapter 5, "Communication", describes, transaction-level interfaces for transactors and mechanisms for passing transactions/data between transactors such as drivers/monitors and scoreboards. Channel, TLM transport and analysis port and notifications are reviewed here.
- Chapter 6, "Implementing Tests & Scenarios", provides various scenario generation mechanisms including the multiple-stream scenario generator (MSSG) classes and features.
- Chapter 7, "Common Infrastructure and Services", introduces infrastructure and elements of VMM. The `vmm_object` class, message services, factory services as well as hierarchical options usage and configuration setup are discussed here.

- Chapter 8, "Methodology Guide", provides the methodology for extending the VMM standard library.
- Chapter 9, "Optimizing, Debugging and Customizing VMM", provides various coding recommendations to optimize performance and describes embedded system functions in VCS helping VMM transaction debug.
- Chapter 10, "Primers", provides procedural examples for understanding MSSG, class factory service, hierarchical options, RTL configurations and modeling a master transactor.
- Appendix A, "Standard Library Classes (Part 1)" and Appendix C, "Command Line Reference" include references to standard library classes and command line switches.

## Basic Concepts of VMM

VMM includes a proven industry-standard verification methodology based on an object-oriented programming model supported by SystemVerilog.

VMM class library provides common infrastructure and services which enable a quick start in building an advanced verification environment. It provides application packages for improving productivity. Using a well-defined and easily accessible library such as VMM, guarantees interoperability of verification components and environments from different sources.

## Building Blocks - Class Library

This section provides an overview of the main VMM class library and utilities used as building blocks for basic verification components. For a complete list and functionality, see Appendix A, "Standard Library Classes (Part 1)" and Appendix C, "Command Line Reference".

### vmm_object

The `vmm_object` virtual base class is used as the common base class for all VMM classes. Classes derived from `vmm_object` and any VMM base class can form a searchable and named object hierarchy.

For details, see Chapter 7, "Common Infrastructure and Services".

### vmm_data [Transactions/Data model]

The `vmm_data` virtual base class is extended to model transactions. This class includes a set of properties, methods and macros required to deal with transactions for different types of designs. For example: `allocate(), copy(), display()`.

For details on `vmm_data` and transaction modeling, see Chapter 3, "Modeling Transactions".

### vmm_xactor Transactors, such as Drivers, Monitors]

The `vmm_xactor` virtual base class is extended to model all kinds of transactors such as, bus-functional models, monitors and generators. This class includes properties and methods used to configure and control different types of transactors.

For details on `vmm_xactor` class, see Chapter 4, "Modeling Transactors and Timelines".

### `vmm_channel [Communication, Transaction Passing]

The `` `vmm_channel `` class defines a transaction-level interface class that serves as the conduit for transaction exchange between transactors in the verification environment. The channel class includes properties and methods used to control the flow of transactions between transactors. For example: `full_level(), size(), is_full()`.

For details on channel class, see Chapter 5, "Communication".

### vmm_tlm_* [Communication, Transport Interface Mechanisms]

The `vmm_tlm*` classes emulate the following OSCI TLM 2.0 transport interfaces: blocking, non-blocking, socket, generic payload and analysis port.

For details on `vmm_tlm_*` class, see Chapter 5, "Communication".

### vmm_ms_scenario and vmm_ms_scenario_gen

The general purpose MSSG controls, schedules and randomizes multiple stimulus scenarios. Multi-stream scenarios are able to inject stimulus or react to response on multiple channels. You can also create hierarchical scenarios that are composed of other multi-stream scenarios.

For details, see Chapter 6, "Implementing Tests & Scenarios".

### vmm_class_factory [VMM factory service]

The factory service provides a simple set of APIs to replace any kind of object, transaction, scenario, or transactor with a similar object as required by a specific test.

For details on the factory services, see Chapter 7, "Common Infrastructure and Services".

### `vmm_callback

Callbacks are used to incorporate new mechanisms and routines once a verification environment and its components have been developed. Callback routines are registered in the main routines and executed (or called back) at certain user-defined simulation points. The `` `vmm_callback `` macro defines a callback class that contains methods to be executed when registered callbacks are called.

For details on callbacks, see Chapter 4, "Modeling Transactors and Timelines".

---

## Verification Environments and Execution Control Phases

Phasing refers to the overall progression of a simulation. Execution of a simulation is divided into predefined phases. All verification components within an environment are synchronized to the phases so that their actions can be coordinated throughout. VMM supports explicit, implicit, and mixed phasing.

For details, see Chapter 2, "Architecting Verification Environments".

## vmm_group

The `vmm_group` class is extended to create sub-environment and environments with implicit phasing. All transactors instantiated in this environment have their phases automatically called at the appropriate time.

For details, see Chapter 4, "Modeling Transactors and Timelines".

## vmm_consensus

The `vmm_consensus` object offers a well-defined service for collaboration on deciding test completion and ending simulation.

For details, see Chapter 7, "Common Infrastructure and Services".

## vmm_subenv

The `vmm_subenv` virtual base class is extended to create explicitly-phased sub-environments. All transactors and sub-environments instantiated in this environment must have their phase methods explicitly called at the appropriate time.

For details on `vmm_subenv`, see Chapter 2, "Architecting Verification Environments".

## vmm_env

The `vmm_env` virtual base class is extended to create explicitly-phased environments. This class includes a set of predefined methods that correspond to specific simulation phases. All transactors and sub-environments instantiated in this environment must have their phases methods explicitly called at the appropriate time.

For details on `vmm_env`, see Chapter 2, "Architecting Verification Environments".

## Enhanced Verification Performance and Flexibility

VMM provides comprehensive ways of configuring transactors, components and verification environments which aid improving flexibility and performance.

### vmm_test

The `vmm_test` class is extended to implement testcases. It is where tests add scenarios, override factories and modify connections. The `vmm_test` class can be used for standalone tests or for concatenating multiple implicitly-phased tests within a simulation run to improve overall simulation efficiency.

For details, see Chapter 6, "Implementing Tests & Scenarios".

### vmm_opts

The `vmm_opts` object allows to define and set configuration options. Options can be set from the simulator command line, file or within the code itself. These options can be set on a per-instance basis or globally by using regular expressions.

For details, see Chapter 7, "Common Infrastructure and Services".

## Debug and Analysis: Message Service Class and Transaction Debug

Transactors, scoreboards, assertions, environment and testcases use messages to report regular, debug, or error information.

### vmm_log [message service class]

The `vmm_opts` object provides rich set of severity handling utilities and macros for comprehensive reporting, formatting and analysis.

For details, see Chapter 7, "Common Infrastructure and Services".

### Transaction and Environment Debug

Transaction and components include built-in recording facility that enable transaction and environment debugging. The `vmm_data` class members which are registered using shorthand macros are easily viewed on a waveform.

Additional notification status of various components are viewed on the waveform timeline. Additionally, it is possible to determine the level of debug information that is required to be shown.

For details, see Chapter 9, "Optimizing, Debugging and Customizing VMM".

## What's New in VMM?

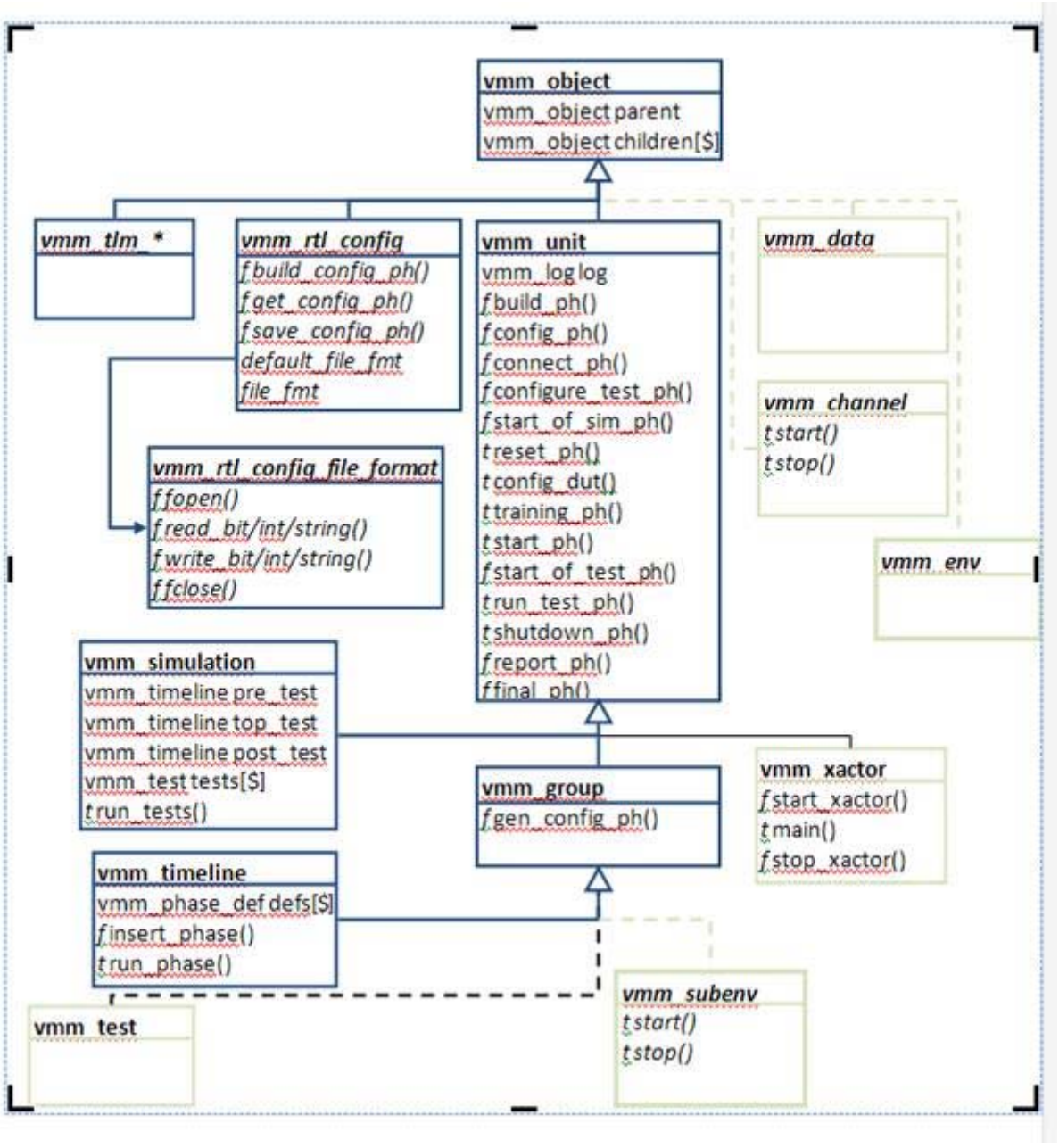
The latest VMM version incorporates new classes and features to enhance the functionality and flexibility in verification environment development.

Some highlights of the new features are,

- The class factory service supplements the existing factory usage and further enables faster stimulus configuration and reuse. It declares and overrides any kind of objects such as, transactions, scenarios, transactors and interfaces.
- The concepts of implicit phasing and timelines for enhanced flexibility and reuse of verification components have been implemented. This augments the current explicit phasing capabilities. Implicit phasing enables components to control their own status.
- Configuration options service including methods to control testbench functionality from the runtime command line, have been enhanced. It supports configuration database and configuration settings from a file. New RTL configuration support ensures alignment with testbench configuration. It supports randomized RTL configuration capabilities.
- Multiple name spaces and hierarchical naming are possible through a new common base class which provides a powerful search functionality.
- TLM-2.0 is now supported, it is complemented with channel-based connectivity and communication mechanisms.
- Extended parameterization features support many base classes in the standard library such as channels and generators.

# UML Diagram

The following diagram shows the relationship between the various VMM classes.



## Resources

The following resources are available for VMM users:

VMM Central (www.vmmcentral.org) is an online community for VMM users to:

- Share information
- Exchange ideas
- Obtain VMM related news and updates
- Receive support on VMM related inquiries
- Learn new tricks and techniques from VMM users and experts

Note: VMM users are strongly encouraged to register as a member. Usage scenarios and recommendations of various VMM features are discussed in the following primers:

- Composing Environments
- Writing Command Layer Master Transactors
- Writing Command Layer Slave Transactors
- Writing Command Layer Monitor Transactors
- Using Command Layer Transactors
- Using the Register Abstraction Layer
- Using the Memory Allocation Manager
- Using the Data Stream Scoreboard

Applications are documented in the following user guides:

- VMM Register Abstraction Layer User Guide
- Verification Planner User Guide
- VMM Hardware Abstraction Layer User Guide
- VMM Scoreboarding User Guide
- VMM Performance Analyzer User Guide

# 2

# Architecting Verification Environments

This chapter contains the following sections:


- “Overview”
- “Testbench Architecture”
- “Constructing and Controlling Environments”
- “Architecting Verification IP (VIP)”
- “Advanced Usage”

## Overview

The challenge in transitioning from a procedural language such as Verilog or VHDL, to a language like SystemVerilog is in making effective use of the object-oriented programming model. When properly used, these features can greatly enhance the reusability of testbench components.

This section covers the following topics:

- Guidelines to maximize the usage of features that create verification components and verification environment satisfying the needs of all the testcases applied to the DUT.
- Guidelines to model transactors with appropriate data sampling interfaces, verification sub-environments and environment.
- Guidelines to model test stimulus and response checking mechanisms.

The guidelines in this chapter are based on the VMM Standard Library specified in Appendix A, "Standard Library Classes (Part 1)". Though the methodology and approaches here can be implemented in a different class library, using a well-defined and openly accessible library guarantees interoperability of the various verification components.

## Testbench Architecture

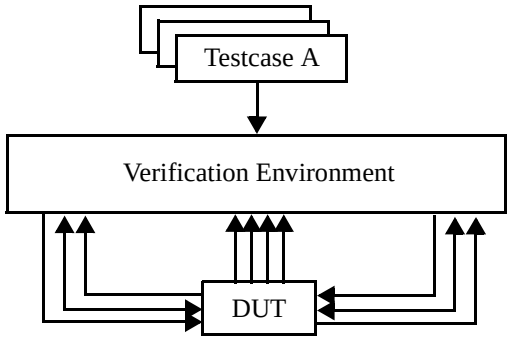
This section describes recommended testbench architecture. You implement testcases on top of a verification environment as shown in Figure 2-1. The verification environment implements the

abstraction and automation functions that help minimize the number and complexity of testcases written. You can reuse the verification environment without modifications with as many testcases as possible to minimize the amount of code required to verify the DUT.

For a given DUT, there might be several verification environments as you can observe in Figure 2-1. However, you should minimize the number of environments and build testcases on top of existing environments as far as possible.

Another important aspect of this methodology is to minimize the number of lines that are required to implement a testcase. Investing in a few or one verification environment to save even a single line in the thousands of potential testcases is worthwhile.

*Figure 2-1 Tests on Top of Verification Environment.*




Verification environments are not monolithic. As shown in Figure 2-2, environments are composed of layers. As in Figure 2-3, they mirror the abstraction layers in the data processed by the design. You design them to meet the various requirements of testcases written for it. Each layer provides a set of services to the upper layers or testcases, while abstracting it from the lower-level details.

*Figure 2-2 Layered Verification Environment Architecture*

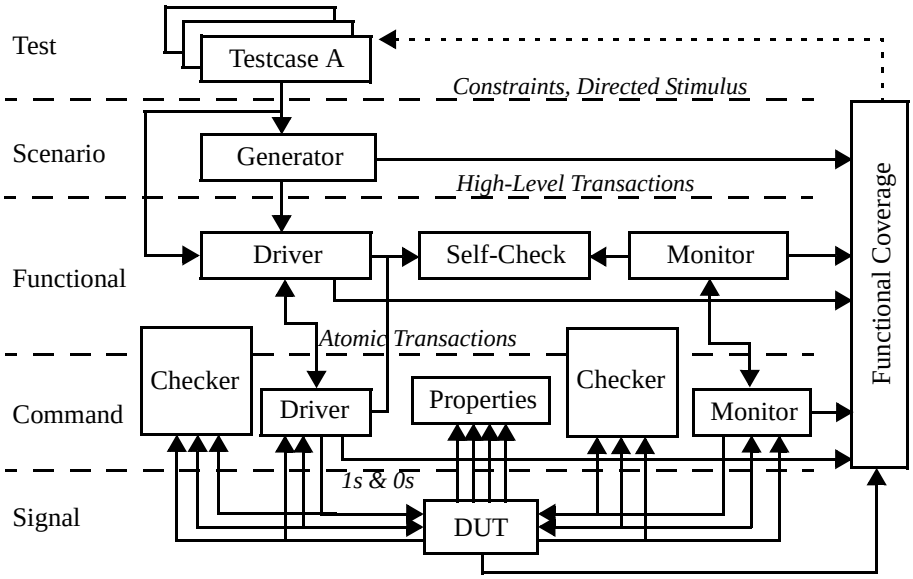
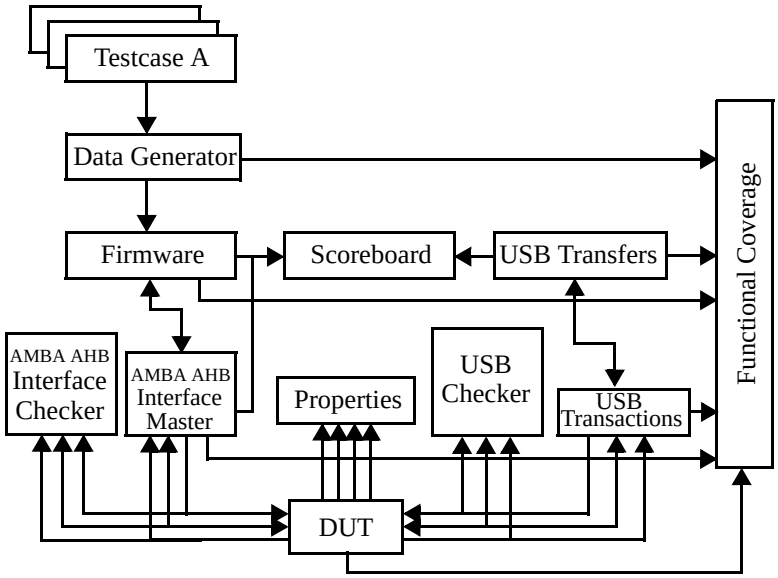



*Figure 2-3 Application of Layered Testbench Architecture.*




Though Figure 2-2 shows testcases interacting only with the upper layers of the verification environment, they can by-pass various layers to interact with various components of the environment or the DUT to accomplish their goals.

Testcases are a combination of additional constraints on generators, new random scenario definitions, synchronization mechanisms between transactors, error injection enablers, DUT state monitoring and directed stimulus.

A verification environment must enable support for all testcases required to verify the DUT without modification. Therefore, you must assemble it with carefully designed, reusable components.

You never implement complete verification environments in one pass. You do not deliver them to the testcase writers as a finished product that implements a complete set of specifications the verification architects provide.

Rather, they evolve to meet the increasingly complex requirements of the testcases being written and responses checked. A trivial directed testcase with no self-checking layers is added to evolve the verification environment into a full-fledged, self-checking, constrained-random one.

The methodology in this chapter allows this evolution to occur in a backward-compatible fashion to avoid breaking existing testcases. It describes enabling the vertical and diagonal reuse of test environments such as block-to-top reuse.

Layered architecture makes no assumption of the DUT model. It can be an RTL, gate-level model or transaction-level model. You can also simulate the DUT natively in the same simulator as the verification environment. Else, co-simulate it on a different simulator or emulate on a hardware platform.

This top-level module contains the design portion of the simulation. Various elements of the signal layer or DUT are accessible via cross-module references through the top-level module. It is unnecessary to instantiate the top-level module anywhere. For guidelines on implementing the signal layer, see “Signal Layer” on page 6.

The environment leverages generic functionality from a verification environment base *`class`*. It refers to the signal layer or various DUT elements via cross-module references into the top-level module.

Each testcase instantiates this environment. For guidelines on implementing the top-level environment class, see “Constructing and Controlling Environments” on page 28.

The `vmm_test` describes the testcase procedure. For guidelines on implementing testcases, see Chapter 6, "Implementing Tests & Scenarios".

## Signal Layer

This layer provides signal-level connectivity to the DUT. Then the signal layer provides pin name abstraction enabling verification components that are used and unmodified with different DUTs or different implementation models of the same DUT. For example, consider an RTL description of the DUT using *`interface`* constructs and a gate-level description of the same DUT using individual bit I/O signals. This layer might abstract synchronization and timing of synchronous signals with respect to a reference signal.

The signal abstraction this layer provides is accessible. All layers and testcases above it might use it where signal-level access is required.

However, you should implement verification environments and testcases in terms of the highest possible level services that lower layers provide and avoid accessing signals directly (unless imperative).

Command-layer transactors have a physical-level interface composed of individual signals. You bundle all signals pertaining to a physical protocol in a single *interface* construct hence allowing this interface to be virtual and easily bound to the DUT. For details, see Chapter 4, "Modeling Transactors and Timelines".

*Example 2-1 Packaging of Interface Declaration*

```
interface mii_if(...);
   ...
endinterface: mii_if;
...
class mii_phy_layer ...;
   virtual mii_if.phy_layer sigs;
   ...
endclass: phy_layer
...
```

If an *interface* declaration already exists for the protocol signals as in RTL design code and it meets (or can be made to meet) all of the subsequent requirements outlined in this section, then you should physically move to the file packaging the transactors that use them. In most cases, different *interface* declarations will exist or you will require them.

To minimize the collisions between *interface* names and other identifiers in the global name space, they use a "likely-unique" prefix. That prefix is the same as various prefixes you use for related transactors. You use the name of the *package* that optionally contains the transactors that use the *interface* as the prefix to further document the association.

Verification components use the same interface constructs regardless of their perspective or role on the interface. Some components drive signals, others simply monitor their value.

Depending on the functionality of the verification component the signal being driven or monitored might be different. Example 2-2 shows how to use *interface* for bundling *inouts* to represent a physical interface signal regardless of the direction of the signal.

*Example 2-2 Verification Interface Signal Declaration*

```
interface mii_if();
   inout       tx_clk;
   inout [3:0] txd;
   inout       tx_en;
   inout       tx_err;
   inout       rx_clk;
   inout [3:0] rxd;
   inout       rx_dv;
   inout       rx_err;
   inout       crs;
   inout       col;
   ...
endinterface: mii_if
```

Example 2-3 shows how to use clocking blocks for modeling synchronous interfaces. This approach avoids race conditions between a design and a verification environment and allows the environment to work with RTL and gate-level models of the DUT without modifications or timing violations.

**You should use parameters to retain default values such as bus width, setup or hold. These values can be overridden when instantiating this interface.**

*Example 2-3 Synchronous Interface Signal Declaration*

```
interface mii_if;
   ...
   parameter setup_time = 5ns;
   parameter hold_time  = 3ns;
```

```
   clocking mtx @(posedge tx_clk);
      default input #setup_time output #hold_time;
      output txd, tx_en, tx_err;
   endclocking: tx

   clocking mrx @(posedge rx_clk);
      default input #setup_time output #hold_time;
      input rxd, rx_dv, rx_err;
   endclocking: rx
   ...
endinterface: mii_if
```

This implementation style allows changing the set-up and hold time on a per instance basis to meet the needs of the DUT without modifying the interface declaration itself. Modifying the interface declaration has global effects. However, you can specify parameters for each interface instance.

*Example 2-4 Specifying Set-Up and Hold Times for Synchronous Signals*

```
mii_if #(.setup_time(1),
         .hold_time  (0)) mii();
```

Different transactors might have different perspectives on a set of signals. One might be a master driver, another a reactive monitor or a slave driver and yet another a passive monitor. Certain interfaces have different types of proactive transactors such as arbiters and agents. You must declare a *modport* for each of their individual perspectives to ensure that each transactor uses the interface signals appropriately.

*Example 2-5 Module Port Declarations*

```
interface mii_if;
   ...
   modport mac_layer(clocking mtx,
                     clocking mrx,
                     input    crs,
                     input    col, ...);
   ...
   modport phy_layer(clocking ptx,
                     clocking prx,
                     output   crs,
```

```
                        output   col, ...);
    ...
    modport passive(clocking ptx,
                    clocking mrx,
                    input crs,
                    input col, ...);
    ...
endinterface: mii_if
```

**You should implement transactors as separate *class* definitions.**

This is described in Chapter 4, "Modeling Transactors and Timelines". They interface to the physical signals through *virtual modports*.

**You should not define transactions and transactors as *tasks* inside the *interface* declaration.**

The *interface* declaration you share with the RTL design might contain such tasks. However, the verification environment uses them.

Note: The signals declared in the interface create a bundle of wires. The direction of information on the individual wires depends on the role of the agent you connect to those wires. For example, wires carrying address information are outputs for a bus master. However, they are inputs for a bus slave or bus monitor.

**You should specify the direction of asynchronous signals directly in the *modport*, for you do not sample them via *clocking* blocks.**

**You should specify the direction of synchronous signals in the *clocking* block and include the entire clocking block in the *modport* port list.**

Thus, synchronous signals are already visible and their directions are already enforced.

Transactors must enable delay of the driving or sampling of synchronous signals by an integer number of cycles. You can specify the number of cycles by referring to the *clocking* block that defines the synchronization of an interface, without knowing the details of the synchronization event specified in the *clocking* block declaration.

Because all signals in a *clocking* block are visible, adding the synchronous signals to the *modport* port list is redundant. Furthermore, referring to synchronous signals through their respective *clocking* blocks highlights their synchronous nature, associated sampling and driving semantics. Example 2-6 shows how to use *clocking* block positive edge for writing BFMs.

*Example 2-6 Waiting for the Next Cycle on the tx Interface*

```
foreach (bytes[i]) begin
   ...
   @(this.sigs.mtx);
   this.sigs.mtx.txd <= nibble;
   ...
   @(this.sigs.mtx);
   this.sigs.mtx.txd <= nibble;
   ...
end
```

You might have written verification components and the design using different *interface* declarations for the same physical signals. To connect the verification components to the design, it is necessary to map two separate *interface* instances to the same physical signals. This can be accomplished with continuous assignments for unidirectional signals and aliasing for bidirectional signals. Example 2-8 shows how to model a top-level module that contains multiple interfaces.

*Example 2-7 Mapping Two Different Interface Instances to the Same Physical Signals*

```
interface eth_tx_if;   // RTL Design Interface
   bit         clk;
   wire  [3:0] d;
   logic       en;
   logic       err;
   logic       crs;
   logic       col;
endinterface: eth_tx_if

module tb_top;

bit        tx_clk;
eth_tx_if  mii_dut(); // Design Interface Instance
mii_if     mii_xct(); // Transactor Interface Instance

assign mii_dut.clk    = tx_clk;  // Unidirectional
assign mii_xct.tx_clk = tx_clk;
alias  mii_xct.txd    = mii_dut.d; // Inout
...
endmodule: tb_top
```

Clock signals must be scheduled in the design regions. Therefore, you must generate them outside the verification environment in an *always* or *initial* block. You should not generate clock signals inside verification components or transactors because they need to be scheduled in the reactive region.

There are race conditions between initial scheduling of the *initial* and *always* blocks implementing the clock generators and those implementing the design.

Delaying the clock edges to a point in time until you have scheduled each *initial* and *always* block at least once, eliminates those race conditions.

It is a good practice to wait for the duration of a few periods of the slowest clock in the system before generating clock edges.

*Example 2-8 Clock Generation in Top-Level Module*

```
module tb_top;
bit tx_clk;
...
initial
begin
   ...
   #20; // No clock edge at T=0
   tx_clk = 0;
   ...
   forever begin
      #(T/2) tx_clk = 1;
      #(T/2) tx_clk = 0;
   end
end
endmodule: tb_top
```

Using a two-state data type ensures that you initialize the clock signals to a known, valid value.

If a four-state logic type such as *logic*, is used to implement the clock signals, the initialization of those signals to *1'b0* might be considered as an active negative edge by some design components.

The alternative of leaving the clock signals at *1'bx* while you delay the clock edges -- as in the previous rule might cause functional problems if you propagate these unknown values.

Clock signals can be synchronized with an asynchronous relationship inherently. This is required to simulate with a fixed initial phase and a common timing reference such as the internal simulation time. You should randomize the relationship of such clocks to ensure that problems related to asynchronous clock domains can surface during simulation.

*Example 2-9 Randomizing Clock Offsets*

```
integer tx_rx_offset;  // 0-99% T lag
integer T = 100;
initial
```

```
begin
   ...
   tx_rx_offset = {$random} % 100;
   #20; // No clock edge at T=0
   tx_clk = 0;
   rx_clk = 0;
   ...
   fork
      begin
         #(T * (tx_rx_offset % 100) / 100.0);
         forever begin
            #(T/2) rx_clk = 1;
            #(T/2) rx_clk = 0;
         end
      end
   join_none

   forever begin
      #(T/2) tx_clk = 1;
      #(T/2) tx_clk = 0;
   end
end
```

To enable tests to control the random clock relationship values, you should randomize random clock relationship values to enable tests to control these values as part of the testcase configuration descriptor. You will then assign to the appropriate variable in the clock generation code, the randomized value in the extension of the explicit *vmm_env::reset_dut()* method or the implicit `reset` phase.

Note: It is possible to pass these values at run-time in the command line by using the `vmm_opts` facility. For details, see Chapter 7, "Common Infrastructure and Services".

## Command Layer

The command layer typically contains bus-functional models, physical-level drivers, monitors and checkers associated with the various interfaces and physical-level protocols present in the DUT.

Regardless of how you model the DUT, the command layer provides a consistent, low-level transaction interface to it. At this level, you define a transaction as an atomic data transfer or command operation on an interface such as a register write, transmission of an Ethernet frame or fetching of an instruction.

You typically define atomic operations using individual timing diagrams in interface specifications. Reading and writing registers is an example of an atomic operation. The command layer provides methods to access registers in the DUT. This layer has a mechanism that bypasses the physical interface to peek and poke the register values directly into the DUT model.

Note: The implementation of direct-access, register read/write driver is dependent upon the implementation of the DUT.

A driver actively supplies stimulus data to the DUT. A *proactive* driver is in control of the initiation and type of the transaction.

Whenever the higher layers of the verification environment supply a new transaction to a proactive driver, the transaction on the physical interface gets immediately executed. For example, a master bus-functional model for an AMBA AHB interface is a proactive driver.

A *reactive* driver is not in control of the initiation or type of the transaction but might be in control of some aspect of the timing of its execution such as the introduction of wait states.

The DUT initiates the transaction and the reactive driver supplies the required data to successfully complete the transaction. For example, a program memory interface bus-functional model is a reactive driver. The DUT initiates read cycles to fetch the next instruction and the bus-functional model supplies new data in the form of an encoded instruction.

A monitor reports observed high-level transaction timing and data information. A *reactive* monitor includes elements to generate the low-level handshaking signals to terminate an interface and successfully complete a transaction.

Unlike a reactive driver, a reactive monitor does not generate transaction-level information. For example, a Utopia Level 1 receiver is a reactive monitor. It receives ATM cells without generating additional data. But it generates a cell to enable signal back to the DUT for flow control.

A *passive* monitor simply observes all signals involved in the transaction without any interference. A passive monitor is suitable for monitoring transactions on an interface between two DUT blocks in a system-level verification environment.

While interfacing with an RTL or gate-level model, the physical abstraction layer might translate transactions to or from signal assertions and transitions.

While interfacing with a transaction-level model, the physical abstraction layer becomes a pass-through layer.

In both cases, the transaction-level interface that is present in the higher layers remains the same. Thereby it allows the same verification environment and testcases to run on different models of the DUT at different levels of abstraction without any modifications.

The services the command layer provides might not be limited to atomic operations on external interfaces around the DUT. You can provide these services on internal interfaces for missing or temporarily removed design components.

For example, embedded memory acting as an elastic buffer for routed data packets can be replaced with a testbench component. This helps track and check packets in and out of the buffer rather than only at DUT endpoints. Or, an embedded code memory in a processor can be replaced with a reactive driver that allows on-the-fly instruction generation instead of using pre-loaded static code. Alternatively, embedded processor can be replaced with a transactor allowing the testbench to control the read and write cycles of the processors, instead of indirectly through code execution.

When replacing DUT components with a transactor, you must take care that it is configured to an equivalent functionality. For example, if the transactor implements a superset of the transactions or timing compared to the DUT component then it should be configured to restrict its functionality to match that of the DUT component.

## Functional Layer

The functional layer provides the necessary abstraction layers to process application-level transactions and verify the correctness of the DUT.

Unlike interface-based transactions of the physical layer, the transactions in the functional layer might not have a one-to-one correspondence with an interface or physical transaction.

Functional transactions are abstractions of the higher-level operations performed by a major subset of the DUT or the entire DUT beyond the physical interface module.

A single functional transaction might require the execution of dozens of command-layer transactions on different interfaces. It depends on the completion status of some physical transactions to retry some transactions or delay others.

Functional layer transactors can be proactive, reactive or passive:

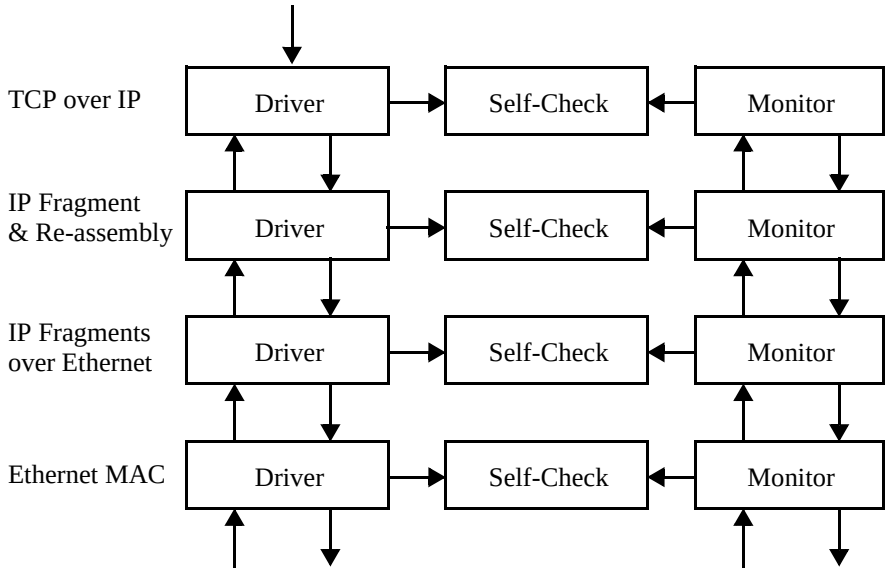
- A proactive transactor controls the initiation and the kind of transaction. It typically supplies some or all of the data the transaction requires.
- A reactive transactor neither controls the initiation nor kind of transaction. It is only responsible for terminating the transaction appropriately by supplying response data or handshaking. Reactive transactors report the observed transaction data they are reacting to.
- Passive transactors monitor transactions on an interface and simply report the observed transactions.

**You should sub-layer the functional layer according to the protocol structure.**

For example, a functional layer for a TCP/IP over Ethernet device should contain a sub-layer to transmit and if necessary, retry an Ethernet frame.

You must provide additional sub-layers to encapsulate IP fragments into Ethernet frames, fragment large IP frames into smaller IP fragments that fit into a single Ethernet frame and encapsulate a TCP packet into an IP frame.

*Figure 2-4 Functional Sub-Layers*




The functional layer is also responsible for configuring the DUT according to a configuration descriptor. This layer includes a functional coverage model for the high-level stimulus and response transactions. It records the relevant information on all transactions this layer processes or creates.

Transactors are implemented using *vmm_xactor*.

*Example 2-10 Modeling Transactor*

```
class mii_phy_layer extends vmm_xactor;
   ...
endclass: mii_phy_layer
...
class tb_env extends vmm_env;
```

```
   ...
   mii_phy_layer phy;
   ...
   virtual function void build();
      ...
      this.phy = new(...);
      ...
   endfunction: build
   ...
   virtual task start();
      ...
      this.phy.start_xactor();
      ...
   endtask: start
endclass: tb_env

program test;
   tb_env env = new;
   ...
endprogram
```

Though different labels are used to refer to stimulus transactors ((driver) from response transactors (monitor), they only differ in the direction of the information flow.

The interfaces on both sets of transactors are transaction-level interfaces. In all other aspects, drivers and monitors operate in the same way and you should implement using the same techniques and offer the same type of capabilities.

## Scenario Layer

This layer provides controllable and synchronizable data and transaction generators. By default, they initiate broad-spectrum stimulus to the DUT. You can use different generators or managers to supply data and transactions at the various sub-layers of the functional layer. This layer also contains a DUT configuration generator.

VMM comes with a general purpose MSSG that aims at controlling, scheduling and randomizing multiple scenarios in parallel. MSSG is a superset of Atomic generators and Scenario Generators. For details, see “Multiple-Stream Scenarios” on page 29.

Atomic generation consists of randomizing individually constrained transactions. Atomic generators are suitable for generating stimulus where putting constraints on sequences of transactions is not necessary. They are suitable for quick randomization bring up and simulation performance.

For example, the configuration description generator is an atomic generator. For details, see “Modeling Scenarios” on page 24.

Scenarios are sequences of random transactions with certain relationships. Each scenario represents an interesting sequence of individual transactions to hit a particular functional corner case.

For example, a scenario in an Ethernet networking operation is a sequence of frames with a specified density i.e., a certain portion of the time the Ethernet line is busy sending/receiving.Otherwise, the line is idle.

MSSG generates scenarios in random order and sequence. It produces a stream of transactions that correspond to the generated scenarios. It initiates scenarios defined by and under the direction of a particular testcase. It produces a stream of transactions that correspond to the requested scenarios.

You might bypass this layer partially or completely by the test layer above it depending on the amount of directedness the testcase requires. Consequently, you must enable turning off generators either from the beginning or in the middle of a simulation to allow the injection of directed stimulus.

**You must enable the restarting of the generator to resume the generation of random stimulus after a directed stimulus sequence.**

Typically, MSGG is a transactor with several transaction-level interfaces and possibly with input interfaces to create scenarios that can react to certain DUT conditions.

As in all other aspects, generators behave like transactors. You should implement them using the same techniques and offer the same type of capabilities.

## Test Layer

Testcases involve a combination of modifying constraints on generators, definition of new random scenarios, synchronization of different transactors and creation of random or directed stimulus.

This layer might provide additional testcase-specific self-checking that is not provided by the functional layer at the transaction level. For example, it checks where correctness will depend on timing with respect to a particular synchronization event introduced by the testcase.

The environment instantiates all necessary transactors and manages their execution. Therefore, the environment that encapsulates them should preferably be instantiated in a *`program`* block.

However, instantiation in *module* is still possible. As an added benefit, the *program* block implementing the testcase is able to access any required element of the verification environment. You instantiate the environment in a local variable to prevent initialization race conditions.

**You should create test by extending the `vmm_test`.**

Example 2-11 shows a simple way of writing test. For details, see “Generating Stimulus” on page 2".

*Example 2-11 Testcase Accessing Verification Environment Elements*

```
program test;
...
tb_env env = new;
initial
begin
   env.run();
end
endprogram: test
```

## Sub-environments

VMM promotes the design of transactors and self-checking structures so that you can reuse them in different environments. For example, you can construct system-level verification environments of the same basic components used to construct block-level environments.

When you construct a system-level environment using the same basic components used to construct block-level environments, VMM arranges, combines and connects these same basic components the same way. For example, a block-level self-checking structure complete with stimulus and response monitors, and scoreboard

might be identical in the system-level environment. This occurs if the system-level, self-checking mechanism, consists of checking the behavior of the individual blocks which compose it.

Similarly, different block-level environments might need similar combinations of basic components. For example, a complete TCP/IP stimulus stack.

You can minimize the overall effort and maintenance if you construct block and system-level environments by reusing complex testbench structures, which already provide a significant portion of the required functionality.

In this section, a "sub-environment" refers to a subset of a verification environment that is reusable in another verification environment. Sub-environments are not individual transactors. They are composed of two or more interconnected transactors potentially linked to additional elements such as, scoreboard, file I/O mechanism or response generator that implement a specific functionality.

You must identify and architect reusable sub-environments in the initial stages while designing and architecting a verification environment. You cannot reuse sub-environments if a verification environment is not designed to take advantage of it.

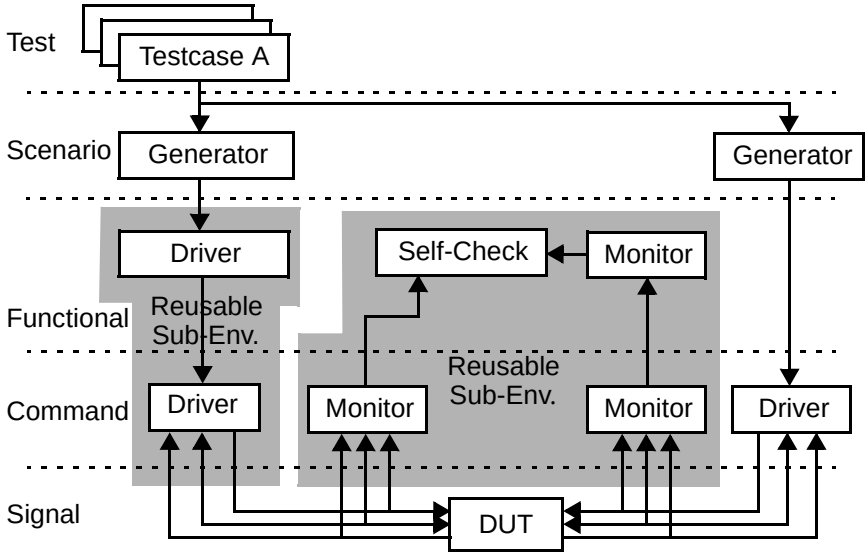
The remainder of this section provides some guidelines and hints to help identify the architect reusable sub-environments.

A sub-environment might span multiple verification environment layers. VMM defines different abstraction layers in verification environments. These layers are more logical than structural. Though

a transactor or basic verification component typically sits in a single layer, a sub-environment can encompass transactors and components in different layers.

For example, Figure 2-5 shows a layered verification environment. The self-checking and stimulus protocol stack structures, which you can make into reusable sub-environments spanning two of those layers.

*Figure 2-5 Layered Verification Environment Architecture*




**A sub-environment might have transaction-level interfaces.**

It is wrong to read too much in Figure 2-5. Though the depicted sub-environments have physical-level interfaces, a reusable sub-environment can also have transaction-level input and outputs, as shown in Figure 2-6.

Physical-level interface are limited to monitoring signal-level activity on a specific physical bus. Transaction-level interface are fed using a different monitor, extracting the same transactions transported on

a different physical bus. They can also be fed from a driver transactor as shown in Figure 2-7, thereby eliminating or delaying the need to develop a command-layer monitor if none is readily available.

*Figure 2-6 Sub-Environment With Transaction-Level Interface*

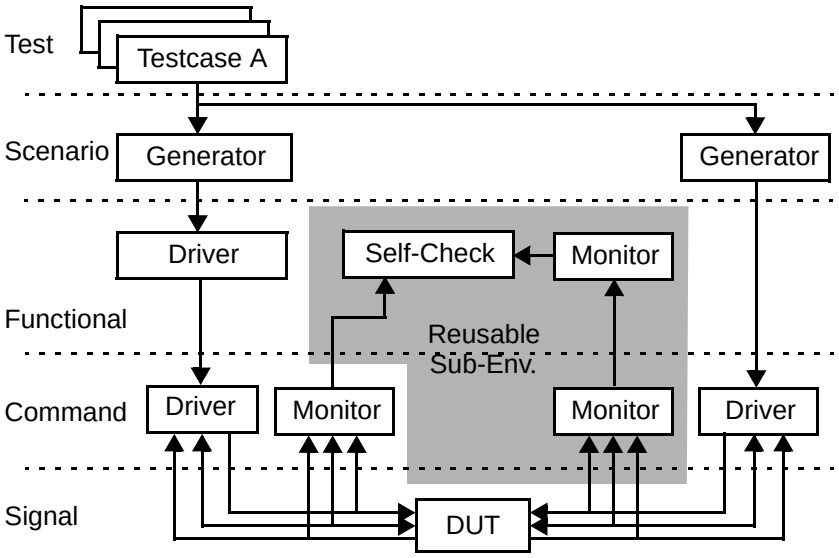
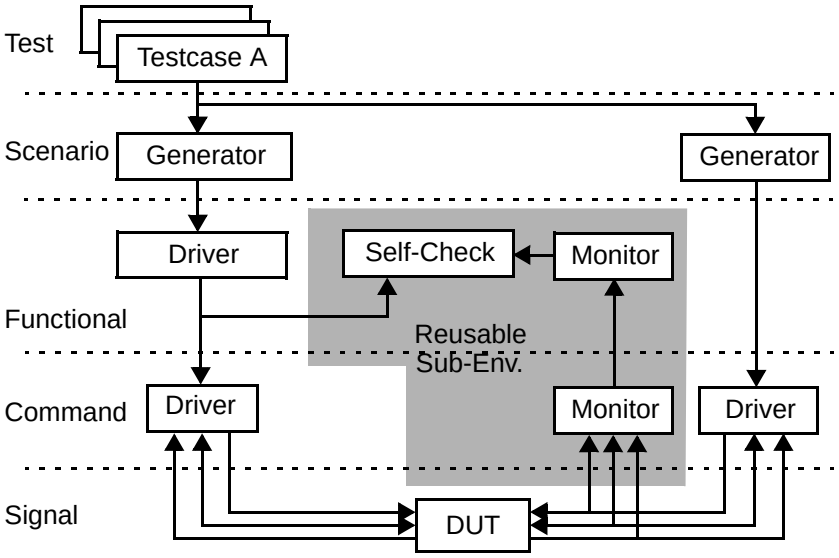


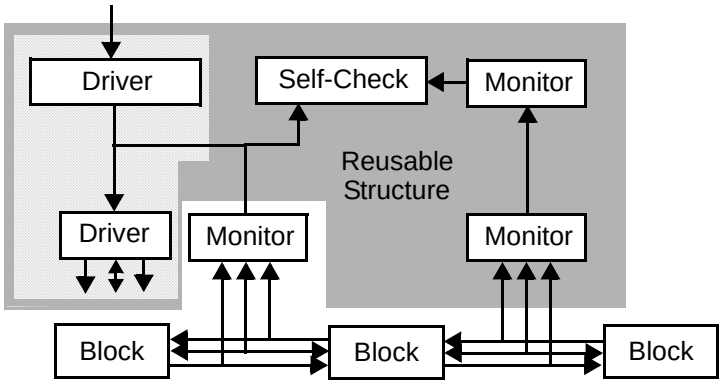
*Figure 2-7 Sub-Environment Interfaced to Driver Transactor*




**The structure of a sub-environment can be configurable.**

Instead of creating two sub-environment as shown in Figure 2-7, you can create a single sub-environment which you can configure with or without the protocol stimulus stack. In a block-level environment, you configure the sub-environment with the protocol stimulus stack. In a system-level environment, another block within the system provides the stimulus. Therefore, you configure the sub-environment without the protocol stimulus stack as shown in Figure 2-8.

*Figure 2-8 Configurable Sub-Environment in System-Level Environment*




There are different ways in which you can specify the configuration of a sub-environment. The following section describes the various techniques.

## Constructing and Controlling Environments

The successful simulation of a testcase to completion involves the execution of the following major functions:

- ***Generating the testcase configuration***. This generation includes a description of the verification environment configuration and the DUT configuration. It also includes a description of the testcase duration. The self-checking feature uses it to determine the appropriate expected response and the verification environment to configure the DUT.

- ***Building the verification environment around the DUT according to the generated testcase configuration***.The used configuration determines the specific type and number of transactors that need to be instantiated around the DUT to exercise it correctly.
  For example, you might configure a DUT with an Intel-style or a Motorola-style processor interface. Each requires a different command-layer transactor. Similarly, you configure 16 GPIO pins as 16 1-bit interfaces or one 16-bit interface (or anything in between). Each configuration requires a different number of command-layer and functional-layer transactors and scoreboards in the self-checking structure:

- ***Disabling all assertions and resetting the DUT***.

- ***Configuring the DUT according to the generated testcase configuration***. This configuration involves writing specific values to registers in the DUT or setting interface pins to specific levels. You should not start transactors and generators as soon as you instantiate them. You must first configure the DUT to be ready to correctly receive any stimulus. Starting the generators too soon complicates the response checking because some initial stimulus sequences must be ignored.

- Enabling assertions and starting all transactors and generators in the environment.

- ***Detecting the end-of-test conditions***. To determine the end of a test using a combination of conditions. Depending on the DUT, testcase terminates after running for a fixed amount of time or number of clock cycles or number of transactions or until a certain number of error messages have been issued or when all monitors are idle.

- Stopping all generators in an orderly fashion

- ***Draining the DUT and collecting statistics***. To determine success of a simulation, it is necessary to drain the DUT of any buffered data or download accounting or statistics registers. Any expected data left in the scoreboard is then assumed to have been lost. Comparison of statistics registers against their expected values is done here.
- ***Reporting on the success or failure of the simulation run***. Not all DUTs require all of those steps. Some steps might be trivial for some DUTs. Others might be very complex. But every successful simulation follows this sequence of generic steps. Individual testcases intervene at various points in the simulation flow to implement the unique aspect of each testcase.

## Quick Transaction Modeling Style

You can easily model transaction with shorthand macros. The only necessary steps are to define all data members and instrument them with macros. Data member macros are type-specific. You must use the macro that corresponds to the type of the data member named in its argument.

**Transaction should be modeled by extending `vmm_data` and using shorthand macros**

*Example 2-12 Transaction Implemented Using Shorthand Macros*

```
class eth_frame extends vmm_data;
  rand bit [47:0] da;
  rand bit [47:0] sa;
  rand bit [15:0] len_typ;
  rand bit [7:0]  data [];
  rand bit [31:0] fcs;
```

```
 `vmm_data_byte_size(1500, this.len_typ + 16)
 `vmm_data_member_begin(eth_frame)
     `vmm_data_member_scalar(da, DO_ALL)
     `vmm_data_member_scalar(sa, DO_ALL)
     `vmm_data_member_scalar(len_typ, DO_ALL)
     `vmm_data_member_scalar_array(data, DO_ALL)
     `vmm_data_member_scalar(fcs,
                             DO_ALL-DO_PACK-DO_UNPACK)
   `vmm_data_member_end(eth_frame)

  constraint valid_frame {
    fcs == 0;
  }
endclass
```

For details, see “Shorthand Macros” on page 24.

## Understanding Implicit and Explicit Phasing

VMM provides two ways of controlling transactor phases from an environment, either implicit or explicit.

- If you want to explicitly call each transactors phase in your environment, simply create an environment that extends `vmm_env`. You need to call the respective transactor phase in the right environment phase. For example, construct transactors in build phase, start transactors in start phase, etc. For details, see “Composing Explicitly Phased Environments” on page 33.

- If you do not want calling each transactor phases at the right time, another modeling style is to use implicit phasing. You need to create an environment that extends `vmm_group`. All transactors that you instantiate in this environment automatically call their phases at the right time. Note that yet it is possible to explicitly call transactor phases herein. For details, see “Composing Implicitly Phased Environments/Sub-Environments” on page 48.

The same statements apply to building sub-environments. For explicitly phased sub-environments, you should extend `vmm_subenv`. For implicitly phased sub-environments, you should extend `vmm_group`.

To decide whether you should use explicitly or implicitly phased, you should consider the following aspects:

- Reuse
  - Implicitly phased sub-environments allow easy vertical reuse from block to system. You can easily remove/add/customize phases.
  - Explicitly phased sub-environments allow easy reuse but you invoke their phases at the right place when you instantiate in environment that extends `vmm_env`.
- ***Ease of Use***
  - Implicitly phased sub-environments are easy to use and you require limited knowledge upon the transactors.
  - Explicitly phased sub-environments phases are well-defined but you should know when and where to add the transactor controls.
- Fine grain control

- Implicitly phased sub-environments control transactor phases automatically. As the time-consuming phases in transactors are divided, it is difficult to control their call order at times.
- Explicitly phased sub-environments phases are well-defined but you should know when and where to add the transactor controls.

Note: It is possible for the environment to deal with implicit and explicit phase i.e. the mixed phases. For details, see “Mixed Phasing” on page 75.

## Composing Explicitly Phased Environments

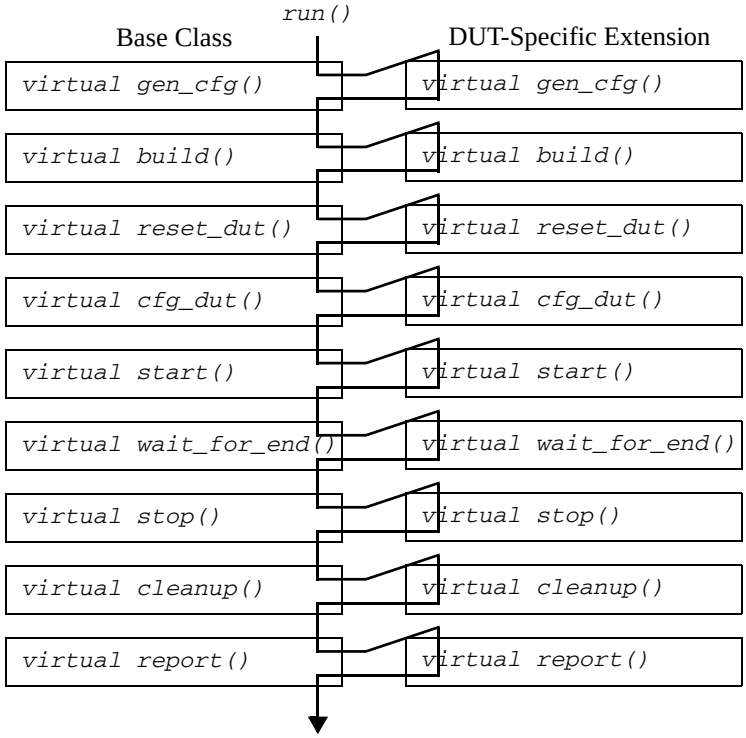
This section describes how to explicitly call each transactors phase in your environment by extending `vmm_env`.

As in Figure 2-9, the *`vmm_env`* base class formalizes these simulation steps into well-defined *virtual methods*. you extend these methods for a verification environment to implement its DUT-specific requirements.

The *`vmm_env`* base class supports the development of a verification environment by extending each virtual method to implement the individual simulation steps the target DUT requires.

The base class already contains the functionality to manage the sequencing and execution of the simulation steps. The DUT-specific environment class extension instantiates and interconnects all transactors, generators and self-checking structures to create a complete layered verification environment around the DUT.

*Figure 2-9 Execution Sequence in vmm_env Class.*




The simulation sequence does not allow a testcase to invoke the *reset_dut()* method in the middle of a simulation, i.e. during the execution of *wait_for_end()*. You should make sure to verify if the design is dynamically reset and reconfigured.

You should implement the body of the *vmm_env::reset_dut()* and *vmm_env::cfg_dut()* in separate tasks. A hardware reset testcase calls these tasks directly to perform the hardware reset and reconfiguration. The reset and reconfiguration sequence is considered part of the *wait_for_end* step for that particular testcase, not a separate step in the simulation.

These methods implement each of the generic steps that must be performed to successfully simulate a testcase.You must overload to perform each step the DUT requires. Even if you don't need to extend a method for a particular DUT, you should extend it anyway and leave it empty to explicitly document that fact.

The implementation of these methods in the base class manages the sequence in which you invoke these methods. They make it unnecessary for each testcase to enumerate all intermediate simulation steps.

Each method extension must call their base implementation first to ensure the proper automatic ordering of the simulation steps. If you violate this rule, the execution sequence of the various simulation steps are broken.

*Example 2-13 Extending Simulation Step Methods*

```
class tb_env extends vmm_env;
   ...
   virtual task wait_for_end();
      super.wait_for_end();
      ...
   endtask
   ...
endclass
```

This method is not virtual because you do not intend to specialize it for a particular verification environment. It is the method that executes the virtual methods in the proper sequence. You must not redefine it to prevent modifying its semantics.

This extension lets tests constrain the testcase configuration descriptor to ensure generation of a desirable configuration without requiring modifications to the environment or configuration descriptor. You can further modify the randomized configuration value procedurally once this method returns

*Example 2-14 Randomization of Testcase Configuration Descriptor*

```
class tb_env extends vmm_env;
   test_cfg cfg;
   ...
   function new();
      super.new();
      this.cfg = test_cfg::create_instance(this,
                                           "Config");

      ...
   endtask

   virtual function void gen_cfg();
      super.gen_cfg();
         if (!this.cfg.randomize()) ...
   endfunction: gen_cfg
   ...
endclass: tb_env
```

The testcase configuration descriptor includes all configurable elements of the DUT and the execution of a testcase. Not only does it describe the various configurable features of the design, but also it includes simulation parameters such as asynchronous clock offsets. It might also include other variable parameters such as, how long to run the simulation for and how many instances of the DUT in the system or the MAC addresses of “known” external devices.

**Configuration object should be constructed as a factory to ensure it can be overridden in your testcases.**

For details, see “Class Factory Service” on page 25.

Some environments do not have any randomizable parameters. Though rare, these environments have a configuration that you describe by an empty configuration descriptor, i.e. a descriptor without any *rand* class properties or descriptor constrained to a single solution.

You must instantiate transactors, generators, scoreboards and functional coverage models according to the testcase configuration. This is typically done in the *vmm_env::build()* method.

The only object you instantiate in the environment constructor is the default testcase configuration descriptor instance that is a randomized in the *vmm_env::gen_cfg()* method extension.

*Example 2-15 Instantiating Environment Components*

```
class tb_env extends vmm_env;
   ...
   function new();
      super.new();
      this.cfg = test_cfg::create_instance(this,
                                           "Config");
      ...
   endtask
   ...
   virtual function void build();
      super.build();
      ...
      this.phy_src = phy_vip::create_instance(this,
                                        "Phy Side", 0);
      ...
   endfunction: build
   ...
endclass: tb_env
```

A testcase should be able to control transactors and generators required to implement its objectives. You can control the transactors and generators directly if they are publicly accessible.

*Example 2-16 Transactor Properties in Verification Environment*

```
class tb_env extends vmm_env;
   ...
   eth_frame_atomic_gen host_src;
   eth_frame_atomic_gen phy_src;
   eth_mac              mac;
   mii_phy_layer        phy;
   ...
endclass: tb_env
```

Integrating the scoreboard into the environment is part of the building process. You can pass to the TLM analysis port if required and all necessary references do exist.

For details, see “Broadcasting Using TLM2.0” on page 31.

*Example 2-17 Integrating Scoreboard Via TLM Analysis Port*

```
class tb_env extends vmm_env;
   ...
   virtual function void build();
      ...
      begin
         sb_mac_sbc sb = new(...);
         // /Bind the MAC analysis port to scoreboard
         this.mac.tlm_bind(sb);
      end
      ...
   endfunction: build
   ...
endclass: tb_env
```

You can use additional analysis port binding to sample data into a functional coverage model or modify the data for error injection. You should ensure that the self-checking structure is aware of all known exceptions or errors injected in the stimulus or observed on the response. This correctly predicts the expected response or assesses the correctness of the observed response.

Some components need access to the transaction to modify or to delay it, before you process it by the transactor. For example, error injection can corrupt a parity byte. You achieve this by registering callback extensions for these particular components.

You should call the callback before you call the analysis port. This ensures that the scoreboard views the actual transaction that is executed.

Because the callback extensions are registered first, these callback extensions are registered using the vmm_xactor::prepend_callback() method.

**Transactor callbacks should be registered in the environment.**

Configuring a DUT often takes a significant amount of simulation time because you usually use a relatively slow processor or serial interface to perform the register and memory updates. Once you verify that interface to ensure that you can have all registers and memories updated, it is no longer necessary to keep exercising that logic.

The DUT-specific extension of the vmm_env::cfg_dut() method should have a “fast-mode” implementation, controlled by a parameter in the testcase configuration descriptor.This causes the performance of all register and memory updates via direct or API accesses, bypassing the normal processor interface.

The environment does not require any additional external intervention to operate properly. You can start all transactors and generators in the extension of the vmm_env::start() method. If a testcase does not require the presence or operation of a particular transactor, you can stop it soon after.

*Example 2-18 Starting Transactors*

```
class tb_env extends vmm_env;
   ...
   virtual task start();
      super.start();
      ...
      this.mac.start_xactor();
      ...
   endtask: start
   ...
endclass: tb_env
```

Configuring the DUT often requires that the configuration and host interface transactors be started in the extension of the vmm_env::cfg_dut() method.

A testcase should be able to control the duration of a simulation. It might be in terms of number of transactions you execute or absolute time or all transactors consenting to stop the simulation.

**An instance of `vmm_consensus` must be present in the environment to terminate test.**

You should use **`vmm_consensus`** blocking task, vmm_consensus::wait_for_consensus() in the vmm_env::wait_for_end() method to control the duration of a simulation.

All contributing components to this consensus should be registered using the `vmm_consensus::register_*()` functions. For details, see “Reaching Consensus for Terminating Simulation” on page 56.

*Example 2-19 Configurable Testcase Duration*

```
class tb_env extends vmm_env;
   vmm_consensus consensus;
   ...
   virtual task wait_for_end();
      super.wait_for_end();
      ...
      concensus.wait_for_consensus();
            ...
   endtask: wait_for_end
   ...
endclass: tb_env
```

## Composing Explicitly Phased Sub-Environments

This section provides guidelines and techniques for implementing reusable sub-environments that you reuse across different verification environments, or instantiate multiple times in the same verification environment.

### You should derive sub-environment classes from `vmm_subenv`.

This base class provides generic functionality required by most sub-environments. It also provides, through virtual methods, standard interfaces for the functionality that the sub-environments must provide.

Furthermore, using a common base class for all sub-environments makes it easy to identify their nature and boundaries. Also, a common base class allows the development of generic functionality to deal with a collection of sub-environments.

For example, an environment can maintain an array of references to all of the sub-environments it contains to easily start and stop all of them.

*Example 2-20 Sub-Environment Declaration*

```
class mii_eth_frame_sb extends vmm_subenv;
   ...
endclass
```

By default, VMM defines this pre-processor symbol to `vmm_subenv`. You might choose to provide your own sub-environment base class derived from the `vmm_subenv` base class. This you do to provide additional organization-specific functionality associated with the particular applications or methods the organization uses.

By redefining the value of the macro from the command line, you can thus derive a sub-environment from an organization-specific base class, even if it comes from outside the organization.

*Example 2-21 Retargetable Sub-Environment Declaration*

```
class mii_eth_frame_sb extends `VMM_SUBENV;
   ...
endclass
```

*Example 2-22 Retargeting Sub-Environment Declarations*

```
% vcs +define+VMM_SUBENV=my_subenv ...
```

**All physical-level interfaces is defined as virtual modport constructor arguments.**

This process documents the physical-level connectivity of the sub-environment. You then directly connect these virtual modports to the appropriate transactors inside the sub-environment.

The following example shows how to pass physical-level interface to a transactor constructor:

*Example 2-23 Physical-Level Interface Definition*

```
function new(   virtual ahb_bus.passive tx_frame,
                virtual mii_phy.passive rx_frame,
            ...);
   ...
endfunction: new
```

**Sub-Environments should have a reference to a configuration descriptor as a constructor argument**

Sub-environments might be configurable in many ways than simply leaving interfaces unconnected. A sub-environment configuration descriptor contains class properties for configuring the sub-environment itself.

Also, the transactors they encapsulate are most likely configurable themselves. A sub-environment configuration descriptor typically contains a configuration descriptor class property for each encapsulated transactor with its own configuration descriptor.

The sub-environment configuration descriptor is typically randomized in the vmm_env::gen_cfg() method extension for the environment containing the reusable structure. You then pass the randomized (or directed) value to the constructor of the sub-environment in the extension of the vmm_env::build() step.

*Example 2-24 Sub-Environment Configuration Descriptor*

```
class mii_eth_frame_sb_cfg;
   rand ahb_cfg ahb;
   rand mii_cfg mii;
endclass: mii_eth_frame_sb_cfg

class mii_eth_frame_sb extends vmm_subenv;
   function new(mii_eth_frame_sb_cfg cfg, ...);
      ...
   endfunction: new
 endclass: mii_eth_frame_sb
```

This process documents the transaction-level connectivity of the sub-environment. You then directly connect these channels to the appropriate transactors inside the sub-environment.

The following examples show how to create a vmm_channel instance in the vmm_env::build() step using a factory.

*Example 2-25 Transaction-Level Interface Definition*

```
class mii_eth_frame_sb extends vmm_subenv;
  eth_frame_channel in_chan;
   function build();
      in_chan = eth_frame_channel::create_instance(this,
                                                   "Chan");
   endfunction
```

```
endclass: mii_eth_frame_sb
```

**Using a task named `configure()` to configure the sub-environment and the portion of the DUT associated with the sub-environment.**

You must configure the sub-environment and the portion of the DUT that corresponds to the functionality it verifies, when the functionality of the sub-environment is configurable.

If the sub-environment and associated DUT functionality are not configurable, this method must still exist to document that fact.

*Example 2-26 Sub-Environment DUT Configuration Method*

```
class mii_eth_frame_sb extends vmm_subenv;
   ...
   task configure(...);
      ...
      super.configured();
   endtask: configure
endclass: mii_eth_frame_sb
```

There is no virtual method in the `vmm_subenv` base class corresponding to this task because it probably requires different arguments for different sub-environments.

The `configure()` method shall call the vmm_subenv::configured() method upon successful completion.

You use the vmm_subenv::configured() method to confirm to the base class that its proper configuration and that of the associated DUT functionality, are done and that it can start.

If you do not invoke this method, the vmm_subenv::do_start() method will issue an error.

**The `configure()` method shall configure the DUT through a register abstraction layer.**

You might reuse a sub-environment associated with a specific block-level DUT in a system-level environment where the corresponding block is no longer directly accessible. The address, physical bus or hierarchical path you use to program registers in the block-level DUT might be different than the ones you use to originally develop the reusable structure.

Registers and memories in a block-level DUT access their current state through a register abstraction layer. This is done regardless of their actual physical context. The appropriate register abstraction interface are then passed as an argument to the *`configure()`* task.

*Example 2-27 Configuring Through Register Abstraction Layer*

```
class mii_eth_frame_sb extends vmm_subenv;
   ...
   task configure(ral_mac_block blk);
      if (this.cfg.mii.duplex) blk.duplex.set(1);
      else blk.duplex.set(0);
      ...
      if (blk.update() != vmm_rw::IS_OK) begin
         ...
         return;
      end
      super.configured();
   endtask: configure
endclass: mii_eth_frame_sb
```

**Extensions of the vmm_subenv implement the vmm_subenv::start(), vmm_subenv::stop() and vmm_subenv::cleanup() virtual methods.**

These methods implement the corresponding generic steps that you must perform to successfully simulate a testcase that includes the sub-environment. You must overload to perform each step the sub-environment requires.

Even if you don't need to extend a method for a particular sub-environment, you should extend it anyway and leave it empty to explicitly document that fact.

You must then call these methods in their corresponding simulation step method in the extension of the `vmm_env` base class where you use a sub-environment.

**Extensions of the, `vmm_subenv::stop()` and `vmm_subenv::cleanup()` virtual methods shall call their base implementation first.**

**The `stop()` method shall stop all registered transactors.**

The implementation of these methods in the base class manages the sequence in which you must invoke these methods. They will report an error if you do not use a sub-environment properly.

*Example 2-28 Extending a Simulation Step Method*

```
class mii_eth_frame_sb extends vmm_subenv;
   ...
   virtual task start();
      super.start();
      this.mii.start_xactor();
      ...
   endtask: start
   ...
enclass: mii_eth_frame_sb
```

**Extensions of the vmm_subenv might implement the `vmm_subenv::report()` virtual method.**

You design this method to implement any status, coverage or statistical reporting of information the sub-environment collects. The default implementation is empty

**Extensions of the `vmm_subenv::report()` method shall not report on the success or failure of the simulation but focus on its registered transactors status.**

You should not use extensions of this method to determine the pass or fail status of the simulation. You should leave this to the `vmm_env::report()` method of the environment instantiating the sub-environment.

If an error is detected that causes the failure of the simulation, it should be reported through a `vmm_log` error message in the `vmm_subenv::cleanup()` method. The message service will record the error message and fail the simulation accordingly.

The sub-environment must be able to participate in the decision of whether or not to end the simulation. This decision must take into account other sub-environments, the overall verification environments and the testcase itself. The `vmm_consensus` utility class offers a well-defined service for collaboration upon deciding when a test is complete and when you can halt the simulation.

**A `vmm_consensus` instance should be available and provided as a reference through the sub-environment constructor.**

How a sub-environment determines if the test can end or not is specific to the sub-environment itself. It can be implemented in various ways:

1. Fork threads in the extension of the `vmm_subenv::start()` method to watch for conditions, such as a generator being done and to consent or disagree to end the test.
2. Have the self-checking structure consent to the end of the test once a pre-determined condition, such as a specific number of observed transactions has been observed.
3. Register all transactors and channels in the sub-environment with the `vmm_consensus` instance to consent to the end of test when all transactors are idle and all channels are empty.

## Composing Implicitly Phased Environments/Sub-Environments

VMM provides the notion of timelines that you use to coordinate the simulation execution for a tree of VMM objects. They are the implicit phasing schedulers.

With implicit phasing, a simulation consists of a series of timelines composed of predefined and user-defined phases. As in Table 2-1, a complete simulation run executes a pre-test timeline, then one or more top-level test timelines and finally a post-test timeline. Each timeline consists of pre-defined phases, which are methods encapsulated in the `vmm_unit` base class.

A timeline controls the phasing of all its children `vmm_unit`. The root `vmm_unit` instances implicitly controls the pre-test, top-level test and post-test timelines. A `vmm_unit` hierarchy might contain sub-timelines at different levels.

Table 2-1 shows the execution order of the predefined phases and their locations within the predefined timelines.

Note: RTL Config phase is defined for `vmm_rtl_config` objects and the others for structural components based on `vmm_unit`.

*Table 2-1 Predefined Phase and Timelines*

| Timeline | Phase | Method |
|---|---|---|
| Pre-test | RTL Config | vmm_rtl_config::* |
| | gen_config | vmm_group::gen_config_ph() |
| | build | vmm_unit::build_ph() |
| | configure | vmm_unit::configure_ph() |
| | connect | vmm_unit::connect_ph() |
| Top-test | configure_test | vmm_unit::configure_test_ph() |
| | start of sim | vmm_unit::start_of_sim_ph() |
| | reset | vmm_unit::reset_ph() |
| | training | vmm_unit::training_ph() |
| | config_dut | vmm_unit::config_dut_ph() |
| | start | vmm_unit::start_ph() |
| | start of test | vmm_unit::start_of_test_ph() |
| | run | vmm_unit::run_ph() |
| | shutdown | vmm_unit::shutdown_ph() |
| | cleanup | vmm_unit::cleanup_ph() |
| | report | vmm_unit::report_ph() |
| Post-test | final | vmm_unit::final_ph() |

The simulation is kick-started by calling `vmm_simulation::run_tests()` from the top-level testbench. The following sequence of phases are called:

- **Step 1: Run the pre-test timeline** on all `vmm_object` hierarchies and selected `vmm_test` instances. The pre-test timeline contains the predefined phases “rtl config”, “gen_config”, “build”, “configure”, and “connect”. This builds, configures and connects the hierarchical verification environment.

- **Step 2: Run the top-level test timeline** for each `vmm_test` instance that are executed in your simulation. The top-level timeline contains the predefined phases, "configure_test", “start_of_sim”, “reset”, “training”, “config_dut”, “start”, “start_of_test”, “run”, “shutdown”, “cleanup” and “report”. You should repeat this step sequentially for every test to be run.
- **Step 3: Execute the post-test timeline**. The post-test timeline contains the predefined phase “final”.

These are the roles of different predefined phases in an environment,

- Pre-test timeline

  This timeline builds, configures and connects the verification environment that will be used by all tests. It is only called once, by the firstly executed test.

  - **RTL configuration**: Create and populate RTL configuration descriptors that reflect the compile-time RTL configuration parameters. You can then use these RTL configuration parameters to affect the structure of the verification environment. For examples, see “RTL Configuration” on page 52.
  - **gen_config:** Perform dynamic configuration of `vmm_group` objects. Registered `vmm_group::gen_config_ph` are called for root objects.
  - **Build**: Instantiate and allocate environment components. You might make VMM channel connections between components optionally here. Registered `vmm_unit::build_ph()` phases are called top down.

- **Configure**: Each component (transactor, generator etc.) provides default configuration values and generates a configuration for the environment either randomly from the top-level in a directed fashion, or using default values in the components. Registered `vmm_unit::configure_ph()` phases are called bottom up.

- **Connect**: Makes connection of Transactor interfaces such as, TLM and Channel here.
  Registered `vmm_unit::connect_ph()` phases are called top down.

• Top-test timeline

  This timeline is the main timeline for the execution of a single test. This is repeated if multiple tests are concatenated in the same simulation run.

  - **Configure_test**: Perform test-specific actions such as, factory replacements, option settings, scenario overrides, callback extensions, etc. When multiple tests are concatenated in the same simulation, each component in the verification environment gets rolled back to this phase. Registered `vmm_unit::configure_test_ph()` phases are called bottom up.

  - **Start_of_sim**: The additional phase you call prior to starting simulation. Registered `vmm_unit::start_of_sim_ph()` phases are called top down.

  - **Reset**: Perform DUT reset, which is typically an activity of the top-level environment. However, in some situations such as, low-power mode/testcase, there are multiple resets happening on different interfaces, in which case the lower level components might implement some functionality in this phase. Registered `vmm_unit::reset_ph()` phases are forked off.

- **Training**: Some interfaces/lower level components require a training phase, this is typically required for reconfiguring transactors based upon DUT parameters, such as timing for a DDR interface, USB low or high speed, etc.
  Registered `vmm_unit::training_ph()` phases are forked off.

- **Config_dut**: Configures DUT through RAL and possibly multi-stream scenarios.
  Registered `vmm_unit::config_dut_ph()` phases are forked off.

- **Start**: Start any execution threads, such as generators, transactors, etc.
  Registered `vmm_unit::start_ph()` phases are forked off.

- **Start_of_test**: Registered `vmm_unit::start_of_test_ph()` phases are called top down.

- **Run**: Termination conditions are watched for completion of the test in this phase.This phase should terminate when all forked threads are done and all `vmm_group` hierarchies consent to the end-of-test.

- **Shutdown**: Optionally stops execution threads. Usually, you need to stop only the generators. Let in-progress data drain from the DUT. Registered `vmm_unit::shutdown_ph()` phases are forked off.

- **Cleanup**: Perform post-tests checking operations, such as, reading accounting registers and checking for orphaned expected responses in the scoreboard. Registered `vmm_unit::cleanup_ph()` phases are forked off.

- **Report**: Perform a pass/fail report for the test. Registered `vmm_unit::report_ph()` phases are called bottom up.

- Post-test timeline
  - **Final**: Perform a final summary report and any action specific to the verification environment such as ensuring the scoreboard is empty, look into coverage bins, etc. Registered `vmm_unit::final_ph()` phases are called bottom up

The pre-test timeline phases in `vmm_group` are implicitly called top down, and thus the phases for the sub-units of the environment will be automatically called in the correct order.

Task-based phases are forked off and must all return for the phase to complete. Execution threads that must survive across phases should be forked off. For details, see “Threads and Processes Versus Phases” on page 18.

## Creating an Implicitly Phased Environment

Implicitly phased environment usually contains the various testbench components such as transactors, monitors, generators, coverage model, etc.

**You should implement implicitly phased environment by extending the `vmm_group` base class, not the `vmm_unit` as it is virtual.**

Example 2-29 describes an implicitly phased environment containing one transactor and one generator. It demonstrates how to instantiate them, connect them using a channel, and implement a few relevant phases.

*Example 2-29 Instantiating VIP in Implicitly Phased Environment*

```
`include “vip_trans.sv”
class my_env extends vmm_group;
   `vmm_typename(my_env)
```

```
    vip  bfm1;
    gen  gen1;

    function new(string inst="", vmm_unit parent = null);
       super.new("my_env", inst, parent);
    endfunction

    virtual function void build_ph();
       bfm1 = new(this, "bfm1");
       gen1 = new(this, "gen1");
    endfunction

    function void configure_ph();
      `vmm_note(log, "configure_ph...");
      // override default configuration for the environment
      vmm_opts::set_int("bfm1:param",2);
    endfunction

    function void connect_ph(); //connect components
      `vmm_note(log, "connect_ph...");
      vmm_connect#(vip_trans_chan)::channel(gen1.out_chan,
                                            bfm1.in_chan);
    endfunction

   task reset_ph(); //Device specific reset
     `vmm_note(log, " reset...");
   endtask

   task config_dut_ph(); //Implement DUT
                       // initialization sequences
      `vmm_note(log, " config_dut...");
      // Drive directed sequences, or a specific
      // initialization scenario from MSS generator
   endtask

   task shutdown_ph(); //wait-till-end-of-test consensus
      `vmm_note(log, " All children signal completion...");
   endtask
endclass: my_env
```

## Completing the “run” Phase

The "run" phase is the place where you perform the main part of the test.

Each `vmm_group` instance contains a vmm_consensus instance and provides `consent()` and `oppose()` methods. By default, a unit consents. Furthermore, the `vmm_consensus` of all children `vmm_groups` are registered with their parent consensus, thus creating a hierarchy of consensus whose consent or opposition percolates to the top-level unit. For details, see “Reaching Consensus for Terminating Simulation” on page 56

A timeline will remain in the "run" phase until:

- All forked off `vmm_group::run_ph()` tasks terminate
- All `vmm_group` instances that reside under the same timeline consent via their `vmm_group::vote vmm_consensus` instance.

Without further actions, all generic voter interfaces consent and test reaches the overall consensus. You can register additional participants, transactors, channels and generic voters. For example, you might register a transactor so it consents only when it is idle.

In Example 2-30, note how the transactor registers with the consensus of its encapsulating `vmm_group::vote` and not with its own `vmm_consensus` instance. This is to allow the user of a transactor to decide whether or not the transactor being idle is a required condition for the end of test.

*Example 2-30 Modeling Implicitly Phased Sub-Environment*

```
class my_subenv extends vmm_group;
   `vmm_typename(my_subenv)
```

```
   my_vip vip1;
   my_vip vip2;

  function new(string name = "", vmm_object parent = null);
      super.new("vip", name, null);
      super.set_parent_object(parent);
   endfunction

   virtual function void build_ph();
      super.build_ph();
      this.vip1 = new(this, "vip1");
      this.vip2 = new(this, "vip2");

   endfunction
   virtual function void connect_ph();
      super.connect_ph();
      this.vote.register_xactor(this.vip1);
      this.vote.register_xactor(this.vip1);
   endfunction
endclass
```

## Reaching Consensus for Terminating Simulation

It is important that a verification environment should decide when to end a test. You design the `vmm_env::wait_for_end()` phase and the completion of the `vmm_simulation::run_tests` method that lets you implement how to detect the end of a test.

After you have identified the end of test, the verification environment can be cleanly shut down. You can also carry out the final accounting of all live stimulus to ensure nothing has been accidentally lost. You can base the termination upon any combination of elapsed time, number of clock cycles, transactors in idle mode or the execution of a pre-defined number of transactions.

When creating a constrained-random verification environment, it is difficult to exactly predict tests duration. Some trivial tests might need to run for only a few transactions, some corner case tests might need to run for several thousand transactions.

Further, in a layered verification environment, it is typically insufficient to count the number of occurrences of a significant event at a single location.

For example, counting the number of packets that are injected in the environment might be erroneous as some transactions might be dropped. For safely terminating a testcase in a layered verification environment, it is usually necessary to wait for a combination of several different conditions:

- All generators have generated the number of required transactions
- All transactors are idled
- No transactions remain in transaction-level interfaces
- Enough activity has been observed by the scoreboard
- The DUT has flushed out remaining transactions

The `vmm_consensus` base class helps identifying when you reach the end of the test for multiple voters. The `vmm_consensus` implements a centralized decision-making mechanism that provides an indication when no participant objects to end this test.

This mechanism is perfectly scalable, allowing verification environments to grow or to combine them without affecting the complexity of the end-of-test decision.

Typically, you shall add an instance of the `vmm_consensus` class to the `vmm_env/vmm_group` class in the `vmm_env::end_vote` property.

The decision to end the test is made by this object. You can distribute contributors to that decision over the entire verification environment. The sum of all contributions helps determine whether to end the test or not, regardless of how many contributors there are. The implementation of the `vmm_env::wait_for_end` step is now only a matter of waiting to reach the end-of-test consensus.

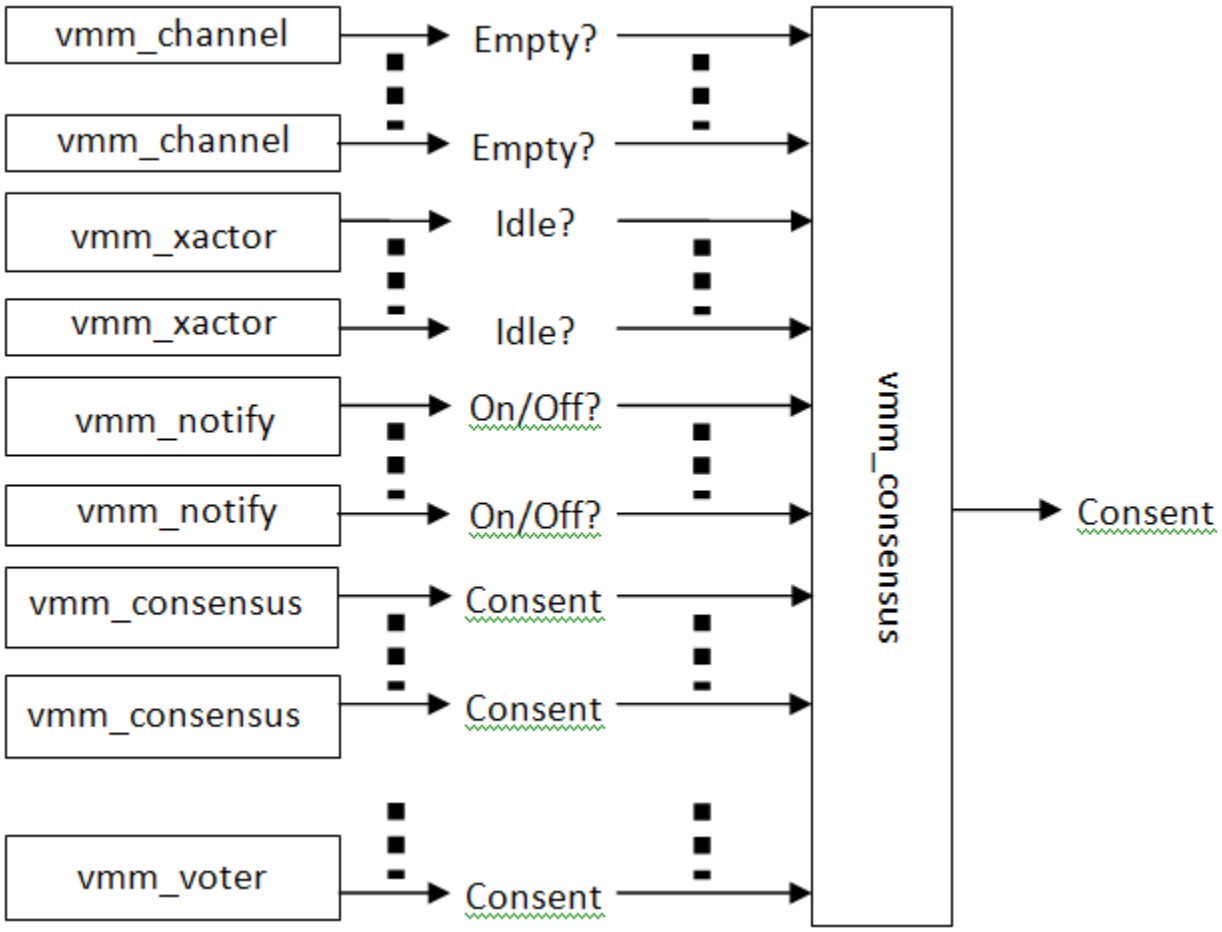
As shown in Figure 2-10, the `vmm_consensus` utility class handles a variety of participants. Each participant can then object or consent to the final decision independent of all other participants.

The following components can be registered as voters:

- Channel instances, which implicitly consent when they are empty.
- Transactor instances, which implicitly consent while they are indicating the `vmm_xactor::XACTOR_IDLE` notification.
- ON/OFF notifications, which implicitly consent while they are indicated (or not).
- Other `vmm_consensus` instances, which implicitly consent when all their own participants consent. This helps creating generic participant interfaces to provide for user-defined agreement or objection to the end-of-test decision.

Using `vmm_consensus`, end-of-test decision process can scale as the complexity of the system-level verification environment increases. It is no longer necessary to implement a complex decision making algorithm with multiple threads watching for different end-of-test conditions. Furthermore, you can pass `vmm_consensus` to sub-environments as you encapsulate it in an object.

*Figure 2-10 Determining End-of-Test Using vmm_consensus*




`vmm_consensus` comes with methods that allow dynamically registering "voters" and blocking until all voters agree to terminate the test.

After registering, voters do not consent for end of test. For example a transactor opposes the end-of-test if it is currently indicating the `vmm_xactor::XACTOR_BUSY` notification. It consents for end of test only when it emits `vmm_xactor::XACTOR_IDLE` notification.

Similarly, a channel opposes until it contains at least one transaction and consents for the end of test when it becomes empty. If you register the VMM notification object, voters consent when it becomes indicated.

`vmm_consensus::wait_for_consensus()` method usually sits in `vmm_env::wait_for_end()` or `vmm_group::run_ph()` method. This method waits until all voters explicitly consent.

The following steps are required to use `vmm_consensus` for terminating a test in an explicitly-phased environment:

- Add `vmm_consensus::wait_for _consensus()` to `vmm_env::wait_for_end()` method
- Add the voters in the `vmm_env::build()` using `vmm_consensus::register_*` method.

Figure 2-31 shows how to terminate a test when all transactors, channels and a voter consent.

*Example 2-31 Scalable End-of-Test using vmm_vote and vmm_consensus*

```
class tb_env extends vmm_env;
  apb_master                          mst;
  apb_slave                           slv;
  apb_trans_atomic_gen                gen;
  apb_sb                              sb;
  vmm_voter                           vote;

function void build();
    apb_trans_channel gen2mst_chan;
    apb_trans_channel mst2slv_chan;
    super.build();

    gen2mst_chan = new("Gen2Mst", "channel");
    mst2slv_chan = new("Mst2Slv", "channel");
    gen = new("ApbGen", 0, gen2mst_chan);
    mst = new("ApbMst", gen2mst_chan, mst2slv_chan);
```

```
    //Add a channel that can participate in this consensus.
     end_vote.register_channel(gen2mst_chan);

     //Add an ON/OFF notification that can
     // participate in this consensus.
     end_vote.register_notification(mst.notify,
                              vmm_xactor::XACTOR_IDLE);

     slv = new("ApbSlv", mst2slv_chan);
     gen.stop_after_n_insts = 10;
     sb = new();
     begin
       sb_master_cbk mcbk = new(sb);
       sb_slave_cbk  scbk = new(sb);
       mst.append_callback(mcbk);
       slv.append_callback(scbk);
     end

     vote = end_vote.register_voter("SB_DONE");
    //Creates a new general-purpose voter interface that
    //can participate in this consensus.
     vote.oppose("XYZ");
   endfunction

task start();
     super.start();
     mst.start_xactor();
     slv.start_xactor();
     gen.start_xactor();

     //Add a transactor that can participate in
     // this consensus.
     end_vote.register_xactor(gen);

     fork begin
        gen.notify.wait_for(apb_trans_atomic_gen::DONE);
       // Create a new voter interface to participate
       // in this consensus
       vote.consent("Generation is done");
     end
     join_none
```

```
endtask

task wait_for_end();
  super.wait_for_end();
  end_vote.wait_for_consensus();
endtask
```

With the new changes of structural components being derived from `vmm_unit` (be it transactors modeled from `vmm_xactor` or environments and sub-senvironments modeled with `vmm_group`), there is a default consensus instance called vote for all these components. By default, this consensus instance consents to simulation completion.

Additionally, two methods are provided: `vmm_unit::oppose`(string "why") and `vmm_unit::consent`(string "why") to explicitly oppose or consent to test completion.

By default, a child 'consensus' instance is registered on the parent's consensus instance. Thus the simulation will complete as `wait_for_consensus` will not be blocking unless one of the components in the hierarchy registers its opposition.

Also, the transactors and environments are now default participants in the 'end of test' detection. This means, you do not necessarily have to call `register_xactor` of the different transactor components. However, a specific invocation of `register_xactor` will ensure that the transactor would consent to a test completion only if the `XACTOR_IDLE` notification is indicated and therefore this continues to be recommended.

For more details on the hierarchical propagation of consents/ opposition to 'end-of-test', see “vmm_unit::request_consensus()” , “vmm_unit::force_thru()” and “vmm_unit::forced()” .

# Architecting Verification IP (VIP)

## VIP and Testbench Components

This section recommends which VMM base class you shall use as the foundation for implementing various elements of a VIP or verification components.

You derive all VMM base classes from the `vmm_object` class. As a result of this implicit parent-child relationship, you can form a searchable, named object hierarchy out of base classes. Also, the constructors of the base classes require a parent and name argument.

The following table summarizes which base class you shall use for specific VIP and testbench components:

*Table 2-2 Base Class Application Summary*

| Application | | Base Class |
|---|---|---|
| Transaction | | vmm_data |
| Transactor | | vmm_xactor |
| Sub-environments | Explicitly phased | vmm_subenv |
| | Implicitly phased | vmm_group |
| Environments | Explicitly phased | vmm_env |
| | Implicitly phased | vmm_group |
| Testcase | | vmm_test |

## Transactions

### Transactions for a specific protocol should extend from **vmm_data**.

For instance, operations such as "read" and "write" of AXI or an ethernet frame of MII could be modeled as transactions. An enumerated-type class property identifies the transaction type and other class properties specify the parameters of the transaction. You declare all such class properties as 'rand'.

You can create default implementations for the `allocate()`, `compare()`, `copy()`, `psdisplay()`, `byte_pack()` and `byte_unpack()` methods by using the `'vmm_data_member_*` shorthand macros.

You shall model transactions using shorthand macros.

The following example shows how you model a simple read/write transaction:

```
class simple_rw extends vmm_data;
   `vmm_typename(simple_rw)

   typedef enum {READ, WRITE} kind_e;
   rand kind_e     kind;
   rand bit [31:0] addr;
   rand bit [31:0] data;
        bit        is_ok;

   `vmm_data_new()
   function new(vmm_object parent = null,
                string      name   = "");
      super.new(parent, name);
   endfunction

   `vmm_data_member_begin(simple_rw)
      `vmm_data_member_enum(kind)
      `vmm_data_member_scalar(addr)
      `vmm_data_member_scalar(data)
      `vmm_data_member_scalar(is_ok)
   `vmm_data_member_end(simple_rw)
```

```
   `vmm_class_factory(simple_rw)
endclass    ...
```

You should always make transaction factory enabled.

This is made possible by simply adding the macro `` `vmm_class_factory `` in the transaction declaration

For every transaction class, a channel transaction-level interface class should be declared:

```
typedef vmm_channel_typed#(simple_rw) simple_rw_channel;
```

## Transactors

Transactors are testbench components that create, execute or observe transactions. Their transaction processing must be started implicitly or explicitly. Transactors can stopped or reset, during which they no longer perform their normal transaction processing.

Transactors have at least one transaction-level input or output interface (using channels or sockets). They might have a physical-level interface. They form the basic elements of a testbench.

Example 2-32 show a simple channel-based master transactor for a simple read/write protocol. The first step is to define transactor callbacks, which are needed for easily grabbing information and modifying this transactor without changing it. For instance, this is useful for injecting errors or add, delay, etc. Here, callback `pre_trans()` is defined and is invoked after the transaction is taken out from the input channel. The other `post_trans()` callback is invoked after the transactor has fully executed the transaction.

Because the callbacks allow to modify the transaction content, they are typically use for injecting errors through transaction or modifying their content.

*Example 2-32 Defining Transactor Callbacks*

```
class master_rw_callbacks extends vmm_xactor_callbacks;
    virtual task pre_trans  (master_rw driver,
                             simple_rw tr,
                             ref bit drop);
    endtask

   virtual task post_trans  (master_rw driver,
                             simple_rw tr
                               );
   endtask
endclass
```

**Transactor should extend the `vmm_xactor` base class.**

In this context, you can instantiate transactor in either an explicitly or implicitly phased environment.

Transactor should contain the following members:

- A virtual interface to drive the DUT signals. Interface usually resides in an object that can be replaced in the command line or anywhere in the environment. Thus making it highly reusable and easy to bind. This only applies to signal-level transactor (for example, a BFM).

**Interface should be replaced in the command line or anywhere in the environment.**

- An input `vmm_channel` to be connected in the connect phase. The transactor does not have to worry about this connection as you usually do it in the environment where you instantiate this transactor.

**Channels are preferred as input connector vs. TLM interfaces**

- One or multiple analysis ports to convey transaction to any subscriber. These ports can be used by the scoreboard or the coverage model.

**You should prefer analysis port to `vmm_notify` for conveying transaction**

**You should use `vmm_notify` for data-less synchronization.**

`callback` to convey transaction that might be modified. As the analysis port does not provide subscribers with the ability to change transaction. You can invoke the analysis port when using a combination of `callback` and analysis port for passing a transaction to other component or subscribers after the callback, to ensure it observes the potentially modified transaction.

**You should use `callback` for transactions that need to be modified**

*Example 2-33 Transactor Declaration*

```
class mastery extends vmm_xactor;
   `vmm_typename(master_rw)

  virtual simple_if.drvprt  iport;
  master_rw_port master_rw_port_obj;
   simple_rw_channel in_chan;

  vmm_tlm_analysis_port#(master_rw, simple_rw)
                          analysis_port;
  ...
endclass
```

Transactor should have a handle to its parent in its constructor arguments.

This is necessary to carry out regular expressions on this particular transactor.

*Example 2-34 Transactor Constructor*

```
function master_rw::new(string inst, vmm_unit parent);
   super.new(get_typename(), inst, 0, parent);
endfunction
```

Transactor should implement the `build` phase that constructs the analysis port and TLM interfaces.

You associate this analysis port with this transactor so that subscribers can trace back to it if necessary. You might need the TLM interfaces for passing transactions in a blocking/non-blocking way.

*Example 2-35 Transactor Build Phase*

```
function void master_rw::build_ph();
  analysis_port = new(this, {get_object_name(),
"_analysis_port"});
endfunction
```

Transactor should implement the `main` thread for implementing its main daemon.

The typical flow of this thread is to,

- Get a transaction from input channel
- Call the appropriate callback, for instance `pre_trans()` callback
- Execute this transaction
- Call the appropriate callback, for instance `post_trans`() callback
- `Write()` this transaction to the analysis port

- Use the appropriate completion model, return status if required
- Possibly stop this flow if the transactor is requested to stop of if the channel is empty. This is achieved by invoking the `vmm_channel::wait_if_stopped_or_empty`() blocking task
- Go to the next transaction

*Example 2-36 Transactor Main Daemon*

```
task master_rw::main();
   bit drop;
   simple_rw   tr;
   is_done = 0;
   fork
     while (1) begin : w0
       this.in_chan.peek(tr);
       if (is_done) break;
      `vmm_trace (this.log, $psprintf ("Driver received a
transaction: %s", tr.psdisplay()));
       `vmm_callback(master_rw_callbacks, pre_trans(this,
tr, drop));

       case (tr.kind)
         simple_rw::READ:
            this.read(tr.addr, tr.data, tr.is_ok);
         simple_rw::WRITE:
            this.write(tr.addr, tr.data, tr.is_ok);
       endcase

      `vmm_callback(master_rw_callbacks, post_trans(this,
tr));
       this.analysis_port.write(tr);
       this.in_chan.get(tr);
       wait_if_stopped();
       if (is_done) break;
     end : w0
   join_none
endtask
```

Transactor should implement the `connect` phase for assigning interfaces.

The purpose is to assign the virtual interface with a configuration which can either be a default one or one specified in the environment with `vmm_opts::set_object_obj()` method.

*Example 2-37 Transactor Connection*

```
function void master_rw::connect_ph();
   bit is_set;
   if ($cast(master_rw_port_obj,
        vmm_opts::get_object_obj(is_set,this,"cpu_port")))
   begin
      if (master_rw_port_obj != null)
         this.iport = master_rw_port_obj.iport;
      else
         `vmm_fatal(log, "Virtual port wrapper not
initialized");
   end
endfunction
```

Transactor should implement the `start_of_sim` phase to ensure channel and interface are present.

*Example 2-38 Transactor Simulation Start*

```
function void master_rw::start_of_sim_ph();
   if (iport == null)
       `vmm_fatal(log, "Virtual port not connected to the
actual interface instance");
endfunction
```

Transactor might implement control phases to gather information on currently executed phases.

Example 2-39 shows how the `shutdown` phase is implemented to assign a status bit whenever the environment timeline reaches the `shutdown` phase.

*Example 2-39 Transactor Shutdown*

```
task master_rw::shutdown_ph();
   is_done = 1;
endtask
```

Transactor should implement operating methods that are necessary to implement the specified protocol and convert the transaction to corresponding DUT pin wiggling.

*Example 2-40 Transactor Operations*

```
   virtual task read(input  bit [31:0] addr,
                     output bit [31:0] data,
                     output bit        is_ok);
      ...
   endtask

   virtual task write(input  bit [31:0] addr,
                      input  bit [31:0] data,
                      output bit        is_ok);
      ...
   endtask
endclass
```

For details, see Modeling Transactors and Timelines Chapter.

## Communication

VMM provides multiple ways of communicating transactors to each other. You can achieve this transaction passing either with `vmm_channel, vmm_tlm, analysis_port` or `callback` interfaces.

Table 2-3 summarizes the interface to use for modeling transactor that issue or receive transactions from other transactors or components.

*Table 2-3 Preferred Communication Media for modeling transactors*

| Base Class | Master | Slave | Monitor |
|---|---|---|---|
| vmm_channel | | Y | |
| vmm_tlm_b_transport | Y | | |
| vmm_analysis_port | Y | Y | Y |
| vmm_callback | Y | Y | |

- Use `vmm_tlm_b_transport` interface for master-like transactor that needs to issue transactions to the other transactor. You can connect this TLM interface to any consumer having either `vmm_channel` or `vmm_tlm_b_transport` interface. Also, `vmm_tlm_b_transport` interface provides a clear completion model that is not tied to the transaction.
- Use `vmm_channel` interface for slave-like transactor that needs to receive transactions from the other transactor.This provides more flexibility as you can connect this channel to any producer like a channel, TLM blocking or non-blocking interface. Furthermore, the `vmm_channel` provides a self-synchronization mechanism that is very handy for slave-like transactor. For example, you can idle this transactor when there are no transactions available in the channel using the `vmm_channel::wait_if_stopped_or_empty()` blocking task.
- Use `analysis_port` for monitor-like transactor that needs to issue an *observed* transaction to other components like scoreboard and coverage models.

## Environments and Sub-Environments

Sub-environments are composed of transactors, other sub-environments and other user-defined classes. Environments are top-level environments.

Table 2-2 provides a summary of the base class to extend for implementing environment or sub-environment.

- To implement an implicitly-phased environment or sub-environment, use `vmm_group` base class.
- To implement an explicit phased sub-environment, use `vmm_subenv` base class.
- To implement an explicitly-phased top-level environment, use the `vmm_env` class.

For details, see “Understanding Implicit and Explicit Phasing” on page 31.

## Testing VIPs

This section gives a brief overview of how to verify your VIP. For details, see Chapter 6, "Implementing Tests & Scenarios".

You implement testcases using the `vmm_test` base class. For details, see “Generating Stimulus” on page 2.

If you write testcases on top of an implicitly-phased top-level environment, you implement them by extending the predefined phasing methods or by defining new phases. For details, see “Understanding Implicit and Explicit Phasing” on page 31.

Example 2-41 shows how to extend the scenario or transaction, add the test-specific constraint and add factory-enabled macro. Extending factory-enabled transaction with test-specific constraints,

*Example 2-41 Extending Test Scenarios*

```
class test_scenario extends simple_rw;
   // Macros to define utility methods
   // like copy/allocate for factory
   `vmm_data_member_begin(test_scenario)
   `vmm_data_member_end(test_scenario)

   constraint cst_dly {
      kind == WRITE;
   }

   // Add factory enable house keeping stuff
   `vmm_class_factory(test_scenario)
endclass
```

As shown in Example 2-42, next action is to extend the `vmm_test`, implement `build` phase to change the number of scenarios to send and override the default generator factory with the test-specific one.

*Example 2-42 Extending vmm_test and Filling in Phases*

```
class test extends vmm_test;
   `vmm_typename(test)

   function new();
      super.new("Test");
   endfunction

  virtual function void build_ph();
     vmm_opts::set_int("%*:num_scenarios", 50);
  endfunction

  virtual function void configure_test_ph();
     simple_rw::override_with_new("@%*",
                     test_scenario::this_type(),
```

```
                    log, `__FILE__, `__LINE__);
  endfunction
endclass
```

If you write testcases on top of an explicitly-phased top-level environment, you implement them by extending the `vmm_test::run()` method and explicitly calling the phase methods in the environment.

```
`vmm_test_begin(test, my_env, "Test")
   env.start();
   `vmm_note(log, "Started...");
   env.run();
   `vmm_note(log, "Stopped...");
`vmm_test_end(test)
```

# Advanced Usage

## Mixed Phasing

It is possible to construct an environment with components using different phasing models.

### Instantiating implicitly phased components in explicitly phased environment

To instantiate one or more implicitly phased components- `vmm_group` or phase-dependent `vmm_xactor` - into an explicitly phased testbench such as `vmm_env` or `vmm_subenv`, they must first be encapsulated under a `vmm_timeline` instance. This timeline can then be explicitly phased by calling the `vmm_timeline::run_phase()` method with the appropriate phase name.

The following examples explain how to use `vmm_timeline` for instantiating implicitly phased sub-environment in explicitly phased environment:

- Example 2-43 shows how to model an implicitly phased sub-environment
- Example 2-44 shows how to instantiate this sub-environment in a timeline
- Example 2-45 shows how to instantiate this timeline in an explicitly phased environment.

*Example 2-43 Implicitly Phased Sub-Environment*

```
class my_subenv extends vmm_group;
   `vmm_typename(my_subenv)

   my_vip vip1;
   my_vip vip2;

  function new(string name = "", vmm_object parent = null);
      super.new("vip", name, null);
      super.set_parent_object(parent);
   endfunction

   virtual function void build_ph();
      super.build_ph();
      this.vip1 = new("vip1", this);
      this.vip2 = new("vip2", this);
   endfunction

   virtual task start_ph();
      super.start_ph();
      `vmm_note(log, "Started...");
   endtask
```

*Example 2-44 Instantiation of Implicitly Phased Sub-Environment in Timeline*

```
class my_tl extends vmm_timeline;
   `vmm_typename(my_tl)

   my_subenv subenv1;

   function new(string name = "", vmm_object parent = null);
      super.new("my_tl", name, parent);
   endfunction

   virtual function void build_ph();
      super.build_ph();
      this.subenv1 = new("subenv1", this);
   endfunction
endclass
```

*Example 2-45 Explicitly Phased Environment With Embedded Timeline*

```
class my_env extends vmm_env;
   `vmm_typename(my_env)

   my_tl tl;

   function new();
      super.new("env");
   endfunction

   virtual function void build();
      super.build();
      this.tl = new("tl", this);
   endfunction

   virtual task start();
      super.start();
      tl.run_phase("start");
      `vmm_note(log, "Started...");
   endtask

   virtual task wait_for_end();
      super.wait_for_end();
```

```
      fork
         tl.run_phase("run");
         begin
            `vmm_note(log, "Running...");
            #100;
         end
      join
   endtask

   virtual task stop();
      super.stop();
      tl.run_phase("shutdown");
      `vmm_note(log, "Stopped...");
   endtask
endclass
```

## Instantiating explicitly phased components in implicitly phased environment

Explicitly phased components such as `vmm_subenv` and `vmm_xactor` can be instantiated directly in implicitly phased components based on `vmm_group`. Their explicit phase control methods, such as `vmm_subenv::start()`, must then be called in an extension of the appropriate phase method in the parent component.

The following examples explain how to instantiate explicitly phased sub-environment in implicitly phased environment:

- Example 2-46 shows how to model an explicitly phased transactor.
- Example 2-47 shows how to model an explicitly phased sub-environment.
- Example 2-48 shows how to instantiate this explicitly phased sub-environment in an implicitly phased environment.

### *Example 2-46 Explicitly Phase Transactor*

```
`include "vmm.sv"

class my_vip extends vmm_xactor;
   `vmm_typename(my_vip)

  function new(vmm_object parent = null, string name = "");
      super.new("vip", name);
      super.set_parent_object(parent);
   endfunction

   virtual function void start_xactor();
      super.start_xactor();
      `vmm_note(log, "Starting...");
   endfunction

   virtual function void stop_xactor();
      super.stop_xactor();
      `vmm_note(log, "Stopping...");
   endfunction
endclass
```

### *Example 2-47 Explicitly Phased Sub-Environment*

```
class my_subenv extends vmm_subenv;
   `vmm_typename(my_subenv)

   my_vip vip1;
   my_vip vip2;

  function new(vmm_object parent = null, string name = "");
      super.new("vip", name, null);
      super.set_parent_object(parent);

      this.vip1 = new(this, "vip1");
      this.vip2 = new(this, "vip2");
   endfunction

   virtual task start();
      super.configured();
      super.start();
```

```
      this.vip1.start_xactor();
      this.vip2.start_xactor();
      `vmm_note(log, "Started...");
   endtask

   virtual task stop();
      super.stop();
      this.vip1.stop_xactor();
      this.vip2.stop_xactor();
      `vmm_note(log, "Stopped...");
   endtask
endclass
```

*Example 2-48 Explicitly Phased Sub-Environment in an Implicitly Phased Testbench*

```
class my_env extends vmm_group;
   `vmm_typename(my_env)

   my_subenv subenv1;

   function new();
      super.new("env");
   endfunction

   virtual function void build_ph();
      super.build_ph();
      this.subenv1 = new(this, "subenv1");
   endfunction

   virtual task start_ph();
      super.start_ph();
      `vmm_note(log, "Started...");
   endtask

   virtual task run_ph();
      super.run_ph();
      `vmm_note(log, "Running...");
      #100;
   endtask
```

```
   virtual task shutdown_ph();
      super.shutdown_ph();
      `vmm_note(log, "Stopped...");
   endtask
endclass
```

# 3

# Modeling Transactions

This chapter contains the following sections:


- “Overview”
- “Class Properties/Data Members”
- “Methods”
- “Factory Service for Transactions”
- “Constraints”
- “Shorthand Macros”

# Overview

The challenge in transitioning from a procedural language such as Verilog or VHDL, to an object-oriented language such as SystemVerilog, is in making effective use of the object-oriented programming model. This section provides guidelines and directives that help modeling transactions by extending `vmm_data`.

Transactions should be modeled by using a `class`, not a `struct` or a `union`. The common tendency is to model transactions as procedure calls such as, `read()` and `write()`. This approach complicates generating random streams of transactions, constraining transactions and registering transactions with a scoreboard.

As shown in Example 3-1, you can model the transactions better by using a transaction descriptor.

*Example 3-1 Transaction Descriptor Object*

```
class wb_cycle extends vmm_data;
   ...
   typedef enum {READ, WRITE, ...} cycle_kinds_e;
   rand cycle_kinds_e kind;
   ...
   rand bit [63:0] addr;
   rand bit [63:0] data;
   rand bit [ 7:0] sel;
   ...
   typedef enum {UNKNOWN, ACK, RTY, ERR,
                 TIMEOUT} status_e;
   status_e status;
   ...
endclass: wb_cycle
```

A transaction is any atomic amount of data eventually or directly processed by the DUT. The packets, instructions, pixels, picture frames, SDH frames and ATM cells are all data items. A data item can be composed of smaller data items by composing a class from smaller classes.

For example, class modeling a picture frame is composed of thousands of instances of a class modeling individual pixels. You can also define all ethernet frame properties in a transaction as shown in Example 3-2.

*Example 3-2 Ethernet MAC Frame Data Model*

```
class eth_frame extends vmm_data;
   ...
   rand bit [47:0] dst;
   rand bit [47:0] src;
   rand bit [15:0] len_typ;
   rand bit [ 7:0] data[];
   rand bit [31:0] fcs;
   ...
endclass: eth_frame
```

The `class` construct has advantages over `struct` or `union` constructs. The latter can model the values contained in the data item only. However, classes can also model operations and transformations such as calculating a CRC value or comparing two instances on these data items using `methods`.

As you assign or copy a reference to the instance, the `class` instances are more efficient to process and move around. The `struct` and `union` instances are scalar variables and you assign and copy their entire content always.

A `class` can also contain constraint definitions to control the randomization of data item values. But `struct` and `union` do not. You can modify the default behavior and constraints of a `class` through `inheritance` without modifying the original base model. `Struct` and `union` do not support `inheritance`.

Tests never call the procedure that implements the transaction, the transactor performs the calling. Instead, tests submit a transaction descriptor to a transactor for execution.

This approach has the following advantages:

- It is easy to create a series of random transactions. Generating random transactions becomes identical to generating random data. You can observe all properties in Example 3-1 having the `rand` attribute.
- You can constrain random transactions. Constraints are applied to object properties only. Constraining transactions that are modeled using procedures requires additional procedural code. You cannot modify procedural constraints such as weights in a `randcase` statement without modifying the source code. Therefore, `randcase` statements prevent reusability.
- You can add new properties to a transaction without modifying its interface. Add them by simply creating a new variant of the transaction object.
- It simplifies integration with the scoreboard. As a transaction is fully described as an object, a simple reference to that object instance passed to the scoreboard, is enough to completely define the stimulus and derive the expected response.

The Chapter 6, "Implementing Tests & Scenarios" shows how you create stimulus using this approach.

# Class Properties/Data Members

This section gives directives for properties and methods used to model, transform or operate on data and transactions.

You must use a static class instance to avoid creating and destroying too many instances of the message service interface as there are thousands of object instances created and destroyed throughout a simulation.

## Quick Transaction Modeling Style

You can easily model transactions with shorthand macros. The only steps required are defining all data members and getting them instrumented with macros. The data member macros are type-specific. You must use the macro that corresponds to the type of the data member named in its argument.

*Example 3-3 Transaction Implemented Using Shorthand Macros*

```
class eth_frame extends vmm_data;
  rand bit [47:0] da;
  rand bit [47:0] sa;
  rand bit [15:0] len_typ;
  rand bit [7:0]  data [];
  rand bit [31:0] fcs;

 ‘vmm_data_byte_size(1500, this.len_typ + 16)
 ‘vmm_data_member_begin(eth_frame)
    ‘vmm_data_member_scalar(da, DO_ALL)
    ‘vmm_data_member_scalar(sa, DO_ALL)
    ‘vmm_data_member_scalar(len_typ, DO_ALL)
    ‘vmm_data_member_scalar_array(data, DO_ALL)
```

```
    `vmm_data_member_scalar(fcs,
                        DO_ALL-DO_PACK-DO_UNPACK)
  `vmm_data_member_end(eth_frame)

  constraint valid_frame {
    fcs == 0;
  }
endclass
```

For details, see “Shorthand Macros” on page 24.

The rest of this section explains how to model transactions and customize data members, methods, etc.

## Message Service in Transaction

Another important aspect of transaction is the ability to issue messages. This is simply done by using Shorthand Macros or explicitly adding a static `vmm_log` instance to the transaction.

*Example 3-4 Declaring and Initializing a Message Service Interface*

```
class eth_frame extends vmm_data;
   static vmm_log log = new("eth_frame", "class");
   ...
   function new();
      super.new(this.log);
   endfunction: new
   ...
endclass: eth_frame
```

Data and transaction descriptors flow through various transactors in the verification environment. Messages related to a particular data object instance are issued through the message service interface in the transactor where there is a need to issue the message.

By this, the location of the message source can be easily identified and controlled. You can include the information on the data or transaction that caused the message in the text of the message or by using the `vmm_data::psdisplay()` method.

## Randomizing Transaction Members

A class must be able to model all possible types of transactions for a particular protocol. You should not use inheritance to describe each individual transaction. Instead, use a class property to identify the type of transaction described by the instance of the transaction descriptor.

If you leave the size of a randomized array unconstrained, it might be randomized to an average length of $2^{30}$. To avoid this situation, you should always constrain the size of a randomized array to a reasonable value. A good practice consists in providing default constraint and constraints that can be externally defined. This practice allows constraints to be turned off or overridden. In Example 3-5, the ethernet frame data payload is constrained to contain less than 1500 elements or 2048 elements is the `valid_len_typ` constraint is turned off.

*Example 3-5 Declaring a class With a Randomized Array*

```
class eth_frame extends vmm_data;
   ...
   rand bit [15:0] len_typ;
   rand bit [ 7:0] data[];
   ...
   constraint valid_len_typ {
      data.size() <= 1500 && len_typ == data.size();
   }
   constraint limit_data_size {
      data.size() < 2048;
   }
   ...
```

```
endclass: eth_frame
```

This approach enables turning off the array's `rand` attributes and constraining them in a derived class, higher-level classes or via the `randomize-with` statement. If the properties are `local`, none of this is possible.

## Context References

Some transactions are layer-based and depend upon lower-level transactions. For instance, the USB protocol comes with high level usb_transfer that consists of a list of usb_packets, which in turn consists of a list of usb_packets. Transaction descriptors for higher-level transactions should have a list of references to the lower-level transactions used to implement them.

You can add lower-level transactors in the verification environment to this list as they implement the higher-layer transaction. The completed list is only valid when the transaction's processing has ended. A scoreboard can then use the list of sub-transactions to determine its status and the expected response.

Conversely, the descriptor for a low-level transaction should have a reference to the higher-level transaction descriptor it implements. This reference helps the scoreboard or other verification environment components to make sense of the transaction and determine the expected response.

In Example 3-6, the higher-layer transaction usb_packet is modeled as a list of usb_transactions, which are modeled as a list of usb_packets.

*Example 3-6 Transaction Context References*

```
class usb_packet extends vmm_data;
```

```
   ...
   usb_transaction context_data;
   ...
endclass: usb_packet

class usb_transaction extends vmm_data;
   ...
   usb_packet packets[];
   usb_transfer context_data;
   ...
endclass: usb_transaction

class usb_transfer extends vmm_data;
   ...
   usb_transaction transactions[];
   vmm_data        context_data;
   ...
endclass: usb_transfer
```

A transaction might be implemented using different lower-level protocols, the implementation references should be of type *vmm_data* to enable reference to any transaction descriptor regardless of the protocol.

Similarly, the context reference of a low-level transaction should be of type *vmm_data* if it should implement or carry information from different higher-level protocols. As shown in Example 3-7, an Ethernet frame can transport any protocol information and should have a generic context reference.

*Example 3-7 Protocol-Generic Context Reference*

```
class eth_frame extends vmm_data;
   ...
   vmm_data context_data;
   ...
endclass: eth_frame
```

## Inheritance and OOP

In traditional object-oriented design practices, inheritance appears to be an obvious implementation. Use a base class for the common properties, then extend it to the various differences in format.

This approach seems the natural choice as the SystemVerilog equivalent to ***e***'s `when` inheritance. Using inheritance to model data formats creates three problems though, two of which are related to randomization and constraints. These are concerns that do not exist in traditional object-oriented languages.

The first problem is the difficulty of generating a stream containing a random mix of different data and transactions formats. This is a requirement for many applications, for example an Ethernet device must be able to accept any mix of various Ethernet frame types on a given port - such as a processor - and must be able to execute any mix of instructions.

Using a common base class gets around the type-checking problem. However, in SystemVerilog, objects must first be instantiated before they can be randomized. Because you must create instances based on their ultimate type, not their base type, the particular format of a data item or transaction is determined before randomizing its content.

Thus, it is impossible to use constraints to control the distribution of the various data and transaction formats or to express constraints on a format as a function of the content of some other class property. For example, if the destination address is equal to this, then the Ethernet frame must have VLAN but no control information.

The second challenge is the difficulty in adding constraints to be applied to all formats of a data item or a transaction descriptor. In adding constraints to a data model, the most flexible mechanism is to create a derived class. Adding a constraint that must apply to all formats of a data model can't be done by simply extending the base class common to all formats. This creates yet another class that is unrelated to the other derivatives. It requires extending each extension of the ultimate class.

The final challenge is that it is not possible to recombine different and orthogonal format variations. For example, the optional VLAN, LLC and control format variations on an Ethernet frame are orthogonal. Hence, there are eight possible variations of the Ethernet frame.

As SystemVerilog does not support multiple inheritance, using inheritance to model this simple case requires eight different classes, i.e. one for each combination of the presence or absence of the optional information. You should solve this problem using proper modeling methodology rather than a new language capability.

Composition is the use of class instances inside another class to form a more complex data or transaction descriptor. Optional information from different formats are modeled by instantiating or excluding a class containing this optional information in the data model. If the information is not present, the reference is set to *null*. Otherwise, the reference would point to an instance containing that information.

This technique has four limitations with randomization:

- Randomization in SystemVerilog does not allocate sub-instances even if the reference class property has the *rand* attribute. Randomization either randomizes a pre-existing instance or does nothing if the reference is *null*.

- It complicates the expression of constraints that might involve a *null* reference. Use constraint guards to detect the absence of optional properties where a *null* reference causes a runtime error.
- It is impossible to express constraints to determine the presence or absence-or their respective ratio-of the sub-instance, it is also impossible to define the data format based on some other (possibly random) properties.
- It brings needless introduction of hierarchies of references to access properties that belong to the same data or transaction descriptor. One must remember whether a class property is optional or not and under which optional instance it is located to access it.

However, a runtime error while attempting to access non-existent information in the current data format is available as a type-checking side effect. But it does not outweigh the other disadvantages.

*Unions* allow multiple data formats to coexist within the same bits. *Tagged unions* enforce strong typing in the interpretation of multiple orthogonal data formats. Unfortunately, tags cannot be randomized. It is impossible to have a *tagged union* randomly select one of the tags or constrain the tag based on other class properties. It is also not possible to constrain fields in randomly-tagged unions because the value of the tag is not defined until solved.

Instead of using inheritance, composition or *tagged unions* to model different data and transaction formats, use the value of a discriminant class property. It is necessary for methods that deal with the ultimate format of the data or transaction such as, *byte_pack()*

and *compare()*.These methods will then procedurally check the value of these discriminant properties to determine the format of the data or transaction and decide on a proper course of action.

*Example 3-8 Using a Discriminant Class Property to Model Data Format*

```
class eth_frame extends vmm_data;
   ...
   typedef enum {UNTAGGED, TAGGED, CONTROL}
      frame_formats_e;
   rand frame_formats_e format;
   ...
   rand bit [47:0] dst;
   rand bit [47:0] src;
   rand bit [ 2:0] user_priority;
   rand bit        cfi;
   rand bit [11:0] vlan_id;
   ...
   virtual function string
      psdisplay(string prefix = "");
      $sformat(psdisplay,
         "%sdst=48'h%h, src=48'h%h, len/typ=16'h%h\n",
         prefix, da, sa, len_typ);
      case (this.format)
      TAGGED: begin
         $sformat(psdisplay,
            "%s%s(tagged) cfi=%b pri=%0d, id=12'h%h\n",
             psdisplay, prefix,
             cfi, user_priority, vlan_id);
      end
      ...
      endcase
      ...
      $sformat(psdisplay, "%s%sFCS = %0s",
               psdisplay, prefix,
               (fcs) ? "BAD" : "good"));
   endfunction: psdisplay
   ...
endclass: eth_frame
```

As you use a single *class* to model all formats, constraints can be specified to apply to all variants of a data type. Use constraints to express relationships between the format of the data and the content of other properties because the format is determined by an explicit class property.

Use constraints to express relationships between the format of the data and the content of other properties. Model orthogonal variations using different discriminant properties, allowing all combinations of variations to occur within a single model.

Inheritance provides for better localization of the various differences in formats but does not reduce the amount of code. It might even increase it. Discriminants might appear verbose but do not require any more lines of code or statements to fully implement.

Furthermore, this technique does not require modeling of optional properties in specific locations amongst other properties to enable some built-in functionality. You implement data and transaction models to facilitate usage, not match some obscure language or simulator requirement.

However, this approach has an apparent disadvantage. There is no type checking to prevent the access of a class property that is not currently valid given the current value of a discriminant class property.

If a strong type checking is required, you can combine this approach with composition to create the data or transaction descriptor. A reference to a subclass that is either *null* or not does not indicate the absence or presence of optional class properties. Instead, the discriminant property indicates that fact.

The descriptor can be fully populated before randomization, then pruned to eliminate the unused class properties. However, it might be difficult to ensure the correct construction of a manually-specified descriptor.

*Example 3-9 Combining a Discriminant Class Property and Composition*

```
class eth_vlan_data;
   rand bit [ 2:0] user_priority;
   rand bit        cfi;
   rand bit [11:0] id;
endclass: eth_vlan_data

class eth_frame extends vmm_data;
   ...
   typedef enum {UNTAGGED, TAGGED, CONTROL}
      frame_formats_e;
   rand frame_formats_e format; // Discrimant
   ...
   rand bit [47:0] dst;
   rand bit [47:0] src;
   rand eth_vlan_data vlan;
   ...
   function void pre_randomize(); // Composition
      if (this.vlan == null) this.vlan = new;
   endfunction

   function void post_randomize();
      if (format != TAGGED) this.vlan = null;
   endfunction
   ...
endclass: eth_frame
```

## Handling Transaction Payloads

Some protocols define fixed fields and data in user-defined payload for certain data types. For example, fixed-format 802.2 link-layer information might be present at the front of the user data payload in

an Ethernet frame. Another example is the management-type frame in 802.11 wherein you replace the content of the user-payload with protocol management information.

You should model the fixed payload data using explicit properties as if they were located in non-user-defined fields. You should reduce the length of the remaining user-defined portion of the payload by the number of bytes used by the fixed payload data, not modeled in explicit properties.

*Example 3-10 Fixed Payload Format Class Property*

```
class eth_frame extends vmm_data;
   ...
   typedef enum {UNTAGGED, TAGGED, CONTROL}
      frame_formats_e;
   rand frame_formats_e format;
   ...
   rand bit [15:0] opcode;
   rand bit [15:0] pause_time;
   ...
   typedef enum [15:0] {PAUSE = 16'h0001} opcodes_e;
   ...
   constraint valid_pause_frame {
      if (format == CONTROL && opcode == PAUSE) begin
          dst     == 48'h0180C2000001;
          max_len == 42;
      end
   }
   ...
   virtual function int unsigned byte_pack(...);
      ...
      case (format)
      ...
      CONTROL: begin
         ... = 16'h8808;
         ... = this.opcode;
         case (this.opcode)
         PAUSE: begin
            ... = this.pause_time;
         end
         endcase
      end
```

```
      ...
    endfunction: byte_pack
...
endclass: eth_frame
```

Data units and transactions often contain information that is optional or unique to a particular type of data or transaction. For example, Ethernet frames might or might not have virtual LAN (VLAN), link-layer control (LLC), sub-network access protocol (SNAP) or control information in any combination. Another example is the instruction set of a processor where different types of instructions use different numbers and modes of operands.

## Methods

This section contains guidelines for using methods in data and transaction models. As explained in previous section, definition and implementation of transaction methods is unnecessary when using shorthand macros. For details, see “Shorthand Macros” on page 24.

You should relate methods in data and transaction descriptors only with their immediate state, i.e. these methods should be non-blocking. There should be no need for advancing the simulation time or suspending the execution thread within these methods. Data and transaction processing requiring advancing time or suspending the execution thread should be located in transactors.

For details, see *vmm_data* base class specification.

These methods provide the basic functionality required to implement a verification environment. They have no built-in equivalent in the SystemVerilog language.

The *vmm_data::allocate()* method is a simple call to *new* and appears redundant. However, it enables the creation of factories and the use of polymorphism in transactors. This is not possible with the direct use of the constructor.

The *vmm_data::copy()* method creates a suitable copy of the data or transaction instance. Whether it is shallow or deep, you should always copy a shallow context references in a descriptor. This method hides the details of the class implementation from you.

It might be necessary to implement these methods if you need to transmit a data model across a physical interface or between different simulations. For example, from SystemVerilog to SystemC.

SystemVerilog does not define *packed* classes. Yet in many instances you must transmit a data item over a certain number of byte lanes across a physical interface. You map back the same stream of data received over the physical interface into higher-level structure and information.

This is automatically handled by *packed struct* and *unions*, but not in classes. The advantages and flexibility of *classes* are unworthy of sacrificing for this simple built-in operation in other data structures. You encapsulate the same functionality in those predefined methods.

The implementation of the *byte_pack()* method shall only pack the relevant properties based on the value of discriminant properties.

Not all properties are valid or relevant under all possible data or transaction formats. The packing methods must check the value of discriminant properties to determine which class property to include in the packed data in addition to their format and ordering. Refer the following example.

Often, discriminant properties are logical properties not directly packed into bit-level data or directly unpacked from it. However, the information necessary to identify a particular variance of a data object is usually present in the packed data. For example, the value *16'h8100* in bytes 12 and 13 of an Ethernet MAC frame stream indicate that the VLAN identification fields are present in the next two bytes. If the information about the data format is unavailable in the bytes to be unpacked, you might use the optional *kind* argument to specify a particular expected format.

The unpacking method must interpret the packed data and set the value of the discriminant properties accordingly. Similarly, it must set all relevant properties to their mapped values based on the interpretation of the packed data. Properties not present in the data stream should be set to unknown or undefined values.

*Example 3-11 Unpacking an Ethernet Frame*

```
class eth_frame extends vmm_data;
   ...
   typedef enum {UNTAGGED, TAGGED, CONTROL}
      frame_formats_e;
   rand frame_formats_e format;
   ...
   rand bit [47:0] dst;
   rand bit [47:0] src;
   rand bit        cfi;
   rand bit [ 2:0] user_priority;
   rand bit [11:0] vlan_id;
   ...
   virtual function int unsigned byte_unpack(
      const ref logic [7:0] array[],
      input int unsigned    offset = 0,
      input int             len    = -1,
      input int             kind   = -1);
      integer i;

      i = offset;
      this.format = UNTAGGED;
      ...
      if ({array[i], array[i+1]} === 16'h8100) begin
```

```
            this.format = TAGGED;
            i += 2;
            ...
            {this.user_priority, this.cfi, this.vlan_id} =
               {array[i], array[i+2]};
            i += 2;
            ...
         end
         ...
      endfunction: byte_unpack
      ...
   endclass: eth_frame
```

You encode the data protection class property simply as being valid or not. Therefore, it must be possible to derive its actual value by other means when necessary.

The method must be virtual to allow the introduction of a different protection value computation algorithm if necessary. When it is modeled as invalid, the packing method is responsible for corrupting the value of a data protection class property, not the computation method. For details, see Example 3-9.

## Factory Service for Transactions

For information, see “Modeling a Transaction to be Factory Enabled” on page 28.

## Constraints

You might model some properties using a type that can yield invalid values. For example, a *length* class property might need to be equal to the number of bytes in a payload array. This constraint ensures that the value of the class property and the size of the array

are consistent. Note that "valid" is not the same thing as "error-free." Validity is a requirement of the descriptor implementation not the data or transaction being described.

*Example 3-12 Basic Frame Validity Constraint Block*

```
class eth_frame extends vmm_data;
   ...
   rand int unsigned min_len;
   rand int unsigned max_len;
   ...
   constraint eth_frame_valid {
      min_len <= max_len;
   }
   ...
endclass: eth_frame
```

Size and duration properties do not have equally interesting values. For example, short or back-to-back and long or drawn-out transactions are more interesting than average transactions.

Randomized class properties modeling size, length, duration or intervals should have a constraint block that distributes their value equally between limit and average values.

*Example 3-13 Constraint Block to Improve Distribution*

```
class eth_frame extends vmm_data;
   ...
   constraint interesting_data_size {
      data.size() dist {min_len                :/ 1,
                        [min_len+1:max_len-1] :/ 1,
                        max_len                :/ 1};
   }
   ...
endclass: eth_frame
```

You should provide a similar specification of value distributions to raise the chances that corner cases are generated.

However, the definition of a corner case is usually DUT-specific. You implement any constraint designed to hit DUT-specific corner cases in a class extension of the data or transaction descriptor, not in the descriptor class itself. This implementation avoids locking in a reusable data or transaction model with DUT-specific information.

*Example 3-14 Adding DUT-Specific Corner-Case Constraints*

```
class long_eth_frame extends eth_frame;
   constraint long_frames {
      data.size() == max_len;
   }
   ...
endclass: long_eth_frame
```

Use one constraint block per class property to make it easy to turn off or override without affecting the distribution of other properties. For details, see Example 3-13.

A conditional constraint block does not imply that the properties used in the expression are solved before the properties in the body of the condition.

If you solve a class property in the body of the condition with a value that implies that the condition cannot be true, this result constrains the value of the properties in the condition. If there is a greater probability of falsifying the condition, it is unlikely to get an even distribution over all discriminant values.

In Example 3-15, if you solve the *length* class property before the *kind* class property, it is unlikely to produce *CONTROL* packets because there is a low probability of you solving the *length* class property as 1.

*Example 3-15 Poor Distribution With Conditional Constraints*

```
class some_packet;
   typedef enum {DATA, CONTROL} kind_typ;
   rand kind_typ kind;
```

```
   rand int unsigned length;
   ...
   constraint valid_length {
      if (kind == CONTROL) length == 1;
   }
endclass: some_packet
```

You can avoid this problem and obtain a better distribution of discriminant properties by forcing the solving of the discriminant class property *before* any dependent class property using the *`solve before`* constraint.

*Example 3-16 Improved Distribution With Conditional Constraints*

```
class some_packet;
   typedef enum {DATA, CONTROL} kind_typ;
   rand kind_typ kind;

   rand int unsigned length;
   ...
   constraint valid_length {
      if (kind == CONTROL) length == 1;
      solve kind before length;
   }
endclass: some_packet
```

You can randomly inject error by selecting an invalid value for error protection properties. A constraint block should keep the value of such properties valid by default. For details, see Example 3-16.

You use one constraint block per error injection class property to make it easy to turn off or override without affecting the correctness of other properties.

You define external *`constraint`* blocks outside the *`class`* that declares them. If you leave them undefined, you consider them empty and do not add constraints to the *`class`* instances. You can define these *`constraint`* blocks later by individual tests to add constraints to all instances of the *`class`*.

*Example 3-17 Declaring Undefined External Constraint Blocks*

```
class eth_frame extends vmm_data;
   ...
   extern constraint test_constraints1;
   extern constraint test_constraints2;
   extern constraint test_constraints3;
   ...
endclass: eth_frame
```

## Shorthand Macros

The implementation of an extension of the `vmm_data` class requires the implementation of many methods. For example, `vmm_data::compare(), vmm_data::copy(), packing, vmm_env::start()`, etc...). Although you only need to implement these methods once, they might be cumbersome to maintain and to implement for trivial class extensions.

However, a set of shorthand macros exist to help reduce the amount of code required to implement or use VMM-compliant data descriptor VMM class extensions. These shorthand macros provide a default implementation of all methods for specified data members.

You specify the shorthand macros inside the class specification after the declaration of the data members. It starts with the `` `vmm_data_member_begin() `` macro and ends with the corresponding `` `vmm_data_member_end() `` macro. In between, you should add corresponding `vmm_*_member_*()` macros for each data member as declared in your transaction.

Data member macros are type-specific. You must use the macro that corresponds to the type of the data member named in its argument.

The order in which you invoke the shorthand data member macros determines the order of printed, compared, copied, packed, and unpacked data members.

*Example 3-18 Transaction Implemented Using Shorthand Macros*

```
class eth_frame extends vmm_data;
   rand bit [47:0] da;
   rand bit [47:0] sa;
   rand bit [15:0] len_typ;
   rand bit [7:0]  data [];
   rand bit [31:0] fcs;

   `vmm_data_byte_size(1500, this.len_typ + 16)
   `vmm_data_member_begin(eth_frame)
      `vmm_data_member_scalar(da, DO_ALL)
      `vmm_data_member_scalar(sa, DO_ALL)
      `vmm_data_member_scalar(len_typ, DO_ALL)
      `vmm_data_member_scalar_array(data, DO_ALL)
      `vmm_data_member_scalar(fcs,
                              DO_ALL-DO_PACK-DO_UNPACK)
   `vmm_data_member_end(eth_frame)

   constraint valid_frame {
      fcs == 0;
   }
endclass
```

Shorthand macros are fully backward compatible with classes implemented using explicitly specified methods. You might choose to implement one class using the shorthand macros and another by explicitly implementing all of the methods.

## User-Defined Implementations

When you use shorthand macros, you provide all `vmm_data` virtual methods with a default implementation. If it is necessary to provide a different, explicitly-coded implementation for one of these methods or data member, you can implement it using one of two approaches.

## User-Defined Method Implementation

If you need a specific implementation for one or two methods, it is recommended to model transactions with shorthand macros and to override these specific methods with your own implementation.

The following methods can be overridden:

```
vmm_data::do_psdisplay()
vmm_data::do_is_valid()
vmm_data::do_allocate()
vmm_data::do_copy()
vmm_data::do_compare()
vmm_data::do_byte_size()
vmm_data::do_max_byte_size()
vmm_data::do_byte_pack()
vmm_data::do_byte_unpack()
```

Example 3-19 shows how to replace the default implementation of the `vmm_data::is_valid()` method by implementing the `vmm_data::do_is_valid()` method. All other methods uses the default implementation provided by shorthand macros.

*Example 3-19 Overloading Default Method Implementation*

```
class eth_frame extends vmm_data;
   rand bit [47:0] da;
   rand bit [47:0] sa;
   rand bit [15:0] len_typ;
   rand bit [7:0]  data [];
   rand bit [31:0] fcs;


   `vmm_data_byte_size(1500, this.len_typ+16)
   `vmm_data_member_begin(eth_frame)
      `vmm_data_member_scalar(da, DO_ALL)
      `vmm_data_member_scalar(sa, DO_ALL)
      `vmm_data_member_scalar(len_typ, DO_ALL)
      `vmm_data_member_scalar_array(data, DO_ALL)
```

```
      `vmm_data_member_scalar(fcs,
                              DO_ALL-DO_PACK-DO_UNPACK)
   `vmm_data_member_end(eth_frame)

   virtual bit function do_is_valid(bit silent = 1,
                                    int kind   = -1);
      if (len_typ < 48)
         return 0;
      if (len_typ < 1500 && len_typ != data.size())
         return 0;
      if (len_typ > 1500 && len_typ < 'h0600)
         return 0;

      return 1;
   endfunction

   constraint valid_frame {
      fcs == 0;
   }
endclass
```

To effectively implement these methods, you must use shorthand macros. However, if you don't use them (for example, you explicitly implement all of the class methods) you must implement the normal, `psdisplay()`,`is_valid()`,`allocate()`,`copy()`,`compare()`, `byte_size()`,`max_byte_size()`,`byte_pack()` and `byte_unpack()` and not their `do_*` counterparts.

## User-Defined Member Default Implementation

If the unsuitable implementation in the default method pertains to a specific data member it is possible to provide a user-defined default implementation for that member.

The user-defined implementation is woven with the other default implementations to create the overall default implementation for all virtual methods.

## User-Defined vmm_data Member Default Implementation

You can provide your own implementation for specific data members. This is possible in conjunction to pre-defined shorthand macros.

You accomplish this for the `vmm_data` class by using the `‘vmm_data_member_user_defined()` macro and implementing a function named `do_<membername>()`. For instance, Example 3-20 provides a specific implementation for `da` member and implements the method called `do_da`.

You **must** implement this method by using the following pattern:

```
function bit do_membername(
     input vmm_data::do_what_e do_what,
     input string prefix,
     ref string image,
     input classname rhs,
     input int kind,
     ref int offset,
     ref logic [7:0] pack1[],
     const ref logic [7:0] unpack1());

   do_name = 1; // Success, abort by returning 0

   case (do_what)
      DO_PRINT: begin
         // Add to the 'image' variable, using 'prefix'
      end
      DO_COPY: begin
         // Copy from 'this' to 'rhs'
      end
      DO_COMPARE: begin
         // Compare 'this' to 'rhs'
         // Put mismatch description in 'image'
         // Returns 0 on mismatch
      end
      DO_PACK: begin
         // Pack into 'pack' starting at 'offset'
```

```
            // Update 'offset' to end of 'pack'
        end
        DO_UNPACK: begin
            // Unpack from 'unpack' starting at 'offset'
            // Update 'offset' to start of next unpacked data
        end
      endcase

endfunction
```

Example 3-20 shows how the default method implementation for the `da` member can be user-specified to display an IP address using the separated hexadecimal value instead of the decimal value provided by the default implementation.

*Example 3-20 User-defined Member Default Implementation*

```
class eth_frame extends vmm_data;
   rand bit [47:0] da;
   rand bit [47:0] sa;
   rand bit [15:0] len_typ;
   rand bit [7:0]  data [];
   rand bit [31:0] fcs;

 `vmm_data_byte_size(1500, this.len+16)
 `vmm_data_member_begin(eth_frame)
      `vmm_data_member_user_defined(da)
      `vmm_data_member_scalar(sa, DO_ALL)
      `vmm_data_member_scalar(len_typ, DO_ALL)
      `vmm_data_member_scalar_array(data, DO_ALL)
      `vmm_data_member_scalar(fcs,
                              DO_ALL-DO_PACK-DO_UNPACK)
   `vmm_data_member_end(eth_frame)

function bit do_da(
    input vmm_data::do_what_e do_what,
    input string prefix,
    ref string image,
    input eth_frame rhs,
    input int kind,
    ref int offset,
    ref logic [7:0] pack1[],
    const ref logic [7:0]unpack1());

   do_da = 1; // Success, abort by returning 0
```

```
   case (do_what)
      DO_PRINT: begin
         $sformat(image, "DA = %h.%h.%h.%h.%h.%h",
                  this.da[47.40], this.da[39.32],
                  this.da[31:24], this.da[23:16],
                  this.da[15: 8], this.da[ 7: 0]);
      end
      DO_COPY: begin
         rhs.da = this.da;
      end
      DO_COMPARE: begin
         if (this.da != rhs.da) begin
             $sformat(image, "this.da (%h.%h.%h.%h.%h.%h)
!= to.da (%h.%h.%h.%h.%h.%h)",
                      this.da[47.40], this.da[39.32],
                      this.da[31:24], this.da[23:16],
                      this.da[15: 8], this.da[ 7: 0],
                      rhs.da[47.40], rhs.da[39.32],
                      rhs.da[31:24], rhs.da[23:16],
                      rhs.da[15: 8], rhs.da[ 7: 0]);
            return 0;
         end
      end
      DO_PACK: begin
         if (pack.size() < offset + 6)
            pack = new [offset + 6] (pack);
        {pack[offset  ], pack[offset+1], pack[offset+2],
         pack[offset+3], pack[offset+4], pack[offset+5]} =
            this.da;
         offset += 6;
      end
      DO_UNPACK: begin
         if (unpack.size() < offset + 6) return 0;
         this.da = {unpack[offset  ], unpack[offset+1],
                    unpack[offset+2], unpack[offset+3],
                    unpack[offset+4], unpack[offset+5]};
         offset += 6;
      end
    endcase

endfunction


   constraint valid_frame {
      fcs == 0;
```

```
   }
endclass
```

Note: You **must** provide a default implementation for all possible operations (`print`, `compare`, `copy`, `pack` and `unpack`).

It is impossible to execute the default implementation that would have otherwise been provided by the other type-specific shorthand macros.

However, it is acceptable to leave the implementation for an operation empty if it is neither going to be used nor has a functional effect.

## Unsupported Data Types

For non-scalar data members, you can provide your own implementation for data members that do not have a pre-defined shorthand macro. For example, a member that is an instance of a user-defined class that is not primarily extended from the `vmm_data` class.

It is necessary that you use the user-defined default member implementation to perform the correct `display`, `copy`, and `compare` operations for that class.

Example 3-21 shows how you can implement `display`, `copy`, and `compare` methods for a user-defined data member called `vlan`.

*Example 3-21 Class Member Default Implementation*

```
class vlan_tag; // no vmm_data extension
   rand bit [ 2:0] pri;
   rand bit        cfi;
   rand bit [11:0] tag;
endclass
```

```
class eth_frame extends vmm_data;
   rand bit [47:0] da;
   rand bit [47:0] sa;
   rand bit [15:0] len_typ;
   rand vlan_tag   vlan;
   rand bit [7:0]  data [];
   rand bit [31:0] fcs;


  `vmm_data_byte_size(1500, this.len+16)
   `vmm_data_member_begin(eth_frame)
      `vmm_data_member_scalar(da, DO_ALL)
      `vmm_data_member_scalar(sa, DO_ALL)
      `vmm_data_member_scalar(len_typ, DO_ALL)
      `vmm_data_member_user_defined(vlan)
      `vmm_data_member_scalar_array(data, DO_ALL)
      `vmm_data_member_scalar(fcs,
                              DO_ALL-DO_PACK-DO_UNPACK)
   `vmm_data_member_end(eth_frame)

function bit do_vlan(
    input vmm_data::do_what_e do_what,
    input string prefix,
    ref string image,
    input eth_frame rhs,
    input int kind,
    ref int offset,
    ref logic [7:0] pack1[],
    const ref logic [7:0] unpack1());

   do_da = 1; // Success, abort by returning 0

   case (do_what)
      DO_PRINT: begin
         if (this.vlan == null) return 1;
         $sformat(image, "%s\n%s   VLAN: %0d/%b (%h)",
                  this.pri, this.cfi, this.tag);
      end
      DO_COPY: begin
         rhs.vlan = (this.vlan == null) ? null
                    : new this.vlan;
      end
      DO_COMPARE: begin
         if (this.vlan == null && rhs.vlan == null)
            return 1;
         if (this.vlan == null) begin
```

```
            image = "No VLAN on this but found on to";
            return 0;
         end
         if (this.rhs == null) begin
            image = "VLAN on this but not on to";
            return 0;
         end
         if (this.vlan.pri != rhs.vlan.pri) begin
        $sformat(image, "this.vlan.pri (%0d) != to.vlan.pri
(%0d)",
                  this.vlan.pri, rhs.vlan.pri);
            return 0;
         end
         if (this.vlan.cfi != rhs.vlan.cfi) begin
         $sformat(image, "this.vlan.cfi (%b) != to.vlan.cfi
(%b)",
                  this.vlan.cfi, rhs.vlan.cfi);
            return 0;
         end
         if (this.vlan.tag != rhs.vlan.tag) begin
         $sformat(image, "this.vlan.tag (%h) != to.vlan.tag
(%h)",
                  this.vlan.tag, rhs.vlan.tag);
            return 0;
         end
      end
      DO_PACK: begin
         if (this.vlan == null) return 1;
         if (pack.size() < offset + 4)
            pack = new [offset + 4 (pack);
         {pack[offset  ], pack[offset+1] = 'h8100};
         {pack[offset+2], pack[offset+3] =}
            {this.vlan.pri, this.vlan.cfi, this.vlan.tag};
         offset += 4;
      end
      DO_UNPACK: begin
         if (unpack.size() < offset + 4) return 1;
         if ({unpack[offset], unpack[offset+1]}
             != 'h8100) return 1;
         this.vlan = new;
         {this.vlan.pri, this.vlan.cfi, this.vlan.tag} =
            {unpack[offset+2], pack[unoffset+3]};
         offset += 4;
      end
   endcase
```

```
endfunction
constraint valid_frame {
      fcs == 0;
}
endclass
```

## rand_mode() copy in Shorthand Macros

The implementation of the `vmm_data::copy()` method provided by the `vmm_data` shorthand macros does not copy the state of the `rand_mode()` for `rand` or `randc` variables.

Therefore, the following new `vmm_data` shorthand macros are defined to copy the `rand_state()` (and only the `rand_state()`) for `rand` or `randc` properties:

- `` `vmm_data_member_rand_scalar(_name, _do) ``
- `` `vmm_data_member_rand_scalar_array(_name, _do) ``
- `` `vmm_data_member_rand_scalar_da(_name, _do) ``
- `` `vmm_data_member_rand_scalar_aa_scalar(_name, _do) ``
- `` `vmm_data_member_rand_scalar_aa_string(_name, _do) ``
- `` `vmm_data_member_rand_enum(_name, _do) ``
- `` `vmm_data_member_rand_enum_array(_name, _do) ``
- `` `vmm_data_member_rand_enum_da(_name, _do) ``
- `` `vmm_data_member_rand_enum_aa_scalar(_name, _do) ``

- `vmm_data_member_rand_enum_aa_string(_name, _do)
- `vmm_data_member_rand_handle(_name, _do)
- `vmm_data_member_rand_handle_array(_name, _do)
- `vmm_data_member_rand_handle_da(_name, _do)
- `vmm_data_member_rand_handle_aa_scalar(_name, _do)
- `vmm_data_member_rand_handle_aa_string(_name, _do)
- `vmm_data_member_rand_vmm_data(_name, _do, _how)
- `vmm_data_member_rand_vmm_data_array(_name, _do, _how)
- `vmm_data_member_rand_vmm_data_da(_name, _do, _how)
- `vmm_data_member_rand_vmm_data_aa_scalar(_name, _do, _how)
- `vmm_data_member_rand_vmm_data_aa_string(_name, _do, _how)

Note:You should use these macros only on `rand` or `randc` properties, else, a syntax error is generated.

The only purpose of these new macros is to copy the `rand_mode()` state. To minimize the run-time performance impact of copying the `rand_mode()` state on large arrays (which must be done on each array element) and on classes with large number of members, the

recommendation is to not use it by default and, if needed, add it in a derived class. It must thus be specified in addition to the non-rand macro to complete the default implementation of the copy method.

```
class vip_tr extends vmm_data;
   rand int huge[65535];

   `vmm_data_member_begin(vip_tr)
      `vmm_data_member_scalar_array(huge, DO_ALL)
   `vmm_data_member_end(vip_tr)
endclass
class directed_tr extends vip_tr;
   function new();
      this.huge.rand_mode(0);
   endfunction

   `vmm_data_new(directed_tr)
   `vmm_data_member_begin(directed_tr)
      `vmm_data_member_rand_scalar_array(huge, DO_ALL)
   `vmm_data_member_end(directed_tr)
endclass
```

# 4

# Modeling Transactors and Timelines

This chapter contains the following sections:


- “Overview”
- “Transactor Phasing”
- “Threads and Processes Versus Phases”
- “Physical-Level Interfaces”
- “Transactor Callbacks”
- “Advanced Usage”

## Overview

The term *transactor* is used to identify components of the verification environment that interface between two levels of abstractions for a particular protocol or to generate protocol transactions.

In Figure 2-2, the boxes labeled Driver, Monitor, Checker and Generator are all transactors. The lifetime of transactors is static to the verification environment. They are created at the beginning of the simulation and stay in existence for the entire duration.

They are structural components of the verification components and they are similar to *modules* in the DUT. Only a handful of transactors get created. In comparison, transactions have a dynamic lifetime. Thousands get created by generators, flow through transactors, get recorded and compared in scoreboards and then freed.

Traditional bus-functional models (BFM) are called *command-layer* transactors. Command-layer transactors have a transaction-level interface on one side and a physical-level interface on the other. Functional-layer and scenario-layer transactors only have transaction interfaces and do not directly interface to physical signals.

This section specifies guidelines designed to implement transactors that are reusable, controllable and extendable. Note that reusability, controllability and extensibility are not goals in themselves.

These features enable reusability of transactors by different testcases and different verification environments. They enable control of transactors to meet the specific needs of the testcases.

You can extend transactors to include the features particular environments require. You must accomplish this control and extension without modifying the transactors themselves to avoid compromising the correctness of known-good transactors and modifying the behavior or functionality of existing testcases.

You may need to use transactors by different verification environments that require different combinations of transactors. Using a unique prefix for all global name-space declarations prevents collisions with other transactors.

*Example 4-1 MII Transactors*

```
class mii_cfg;
   ...
endclass: mii_cfg
...
class mii_mac_layer extends vmm_xactor;
   ...
endclass: mii_mac_layer
...
class mii_phy_layer extends vmm_xactor;
   ...
endclass: mii_phy
```

All declarations a transactor requires must be packaged together. Using a single file to package all these related declarations simplifies the task of bringing all necessary declarations you require to use a transactor in a simulation.

*Example 4-2 Transactors Declarations*

```
class mii_mac_layer extends vmm_xactor;
   ...
endclass: mii_mac
```

Using package-to-package all related declarations might offer the opportunity for separate compilation in some tools. Though a package appears to eliminate the need for a unique prefix, the

potential to use the *"import pkgname::*"* statement still necessitates the clear differentiation of names that might potentially clash in the global name space.

*Example 4-3 MII Transactor Package*

```
package mii;

class mii_cfg extends vmm_data;
   ...
endclass: mii_cfg
...
class mii_mac_layer extends vmm_xactor;
   ...
endclass: mii_mac_layer
...
class mii_phy_layer extends vmm_xactor;
   ...
endclass: mii_phy_layer
...
endpackage: mii
```

Both transactors and data are implemented using the *class* construct. The difference between a transactor *class* and a data *class* is their lifetime. Limited number of transactor instances are created at the beginning of the simulation and they remain in existence throughout. This creates a very large number of data and transaction descriptors instances throughout the simulation and they have a short life span.

Therefore, you can use transactor *classes* like *modules*. Modules instances too, are static throughout the simulation. The current state of each transactor is maintained in local properties and implement the execution threads in local methods. You should use `classes` instead of `modules` because there you perform their instantiation run time. Therefore, the structure of the verification environment you can be dynamically configured according to the generated testcase configuration descriptor.

Modules, being instantiated during the elaboration phase, define a structure before the simulator has had the chance to randomize the testcase configuration descriptor.You also prefer *`classes`* because they offer an implementation protection mechanism. It is possible to limit the access to various properties and methods in the class by declaring them as `protected` or `local`. No such protection mechanism exists in *`modules`*.

The implementation control of the interface that is exposed to you occurs due to protecting the implementation of a *`class`*. And this protection allows the modification of the implementation in a backward-compatible fashion.

With their unrestricted access to all of their internal constructs, *`modules`* might put the implementer in a straitjacket if you use internal state information and procedures.

*`Classes`* also offer the opportunity to provide basic shared functionality to all transactors through a shared base class. Because you do not build *`modules`* on the object-oriented framework, they are not used to offer such shared functionality.

The `vmm_xactor` base class contains standard properties and methods to configure and control transactors. To ensure that all transactors have a consistent usage model, you must derive them from a common base class.

## Transactor Phasing

Transactors progress through a series of phases throughout simulation. All transactors are synchronized so that they execute their phases synchronously with other transactors during simulation execution.

VMM supports two transactor phasing usage: implicit and explicit. In explicit phasing, the transactors are under the control of a master controller such as, `vmm_env` to call the transactor phases. In implicit phasing, the transactors execute their phases automatically and synchronously.

VMM predefines several simulation phases. The following table summarizes these phases and their intended purpose:

*Table 4-1 Predefined VMM Simulation Phases*

| Explicit Phase | Implicit Phase | Intended Purpose |
|---|---|---|
| gen_cfg | rtl_config<br>gen_config | Determine configuration of the testbench |
| build | build | Create the testbench |
| | configure | Configure options |
| | connect | Connect TLM interfaces, channels |
| | configure_test_ph | Test specific changes |
| | start_of_sim | Logical start of simulation |
| reset | reset | Reset DUT |
| | training | Physical interface training |
| cfg_dut | config_dut | Configuration of the DUT |
| start | start | Logical start of test |
| | start_of_test | Physical start of test |
| wait_for_end | run | Body of test, end of test detection to be done here |
| stop | shutdown | Stop flow of stimulus |

*Table 4-1 Predefined VMM Simulation Phases*

| Explicit Phase | Implicit Phase | Intended Purpose |
|---|---|---|
| cleanup | cleanup | Let DUT drain and read final DUT state |
| report | report | Pass/fail report (executed by each test) |
| | final | Final checks and actions before simulation termination (executed by last test only when multiple tests are concatenated) |

## Explicit Transactor Phasing

In explicit phasing, transactors begin to execute when the environment explicitly calls `vmm_xactor::start_xactor` to start the transactor. This then starts the `vmm_xactor::main` thread.

For these functions to work properly, you must fork all threads that implement autonomous behavior for a transactor in the body of the `vmm_xactor::main()` task.

This rule is a corollary of the previous guideline. You cannot control threads started in the constructor by the `vmm_xactor::start_xactor()` and `vmm_xactor::reset_xactor()` methods.

It is important that threads are not started as soon as transactors are instantiated. When the verification environment is initially built and the transactor is instantiated, the DUT might not yet be ready to receive stimulus. Transactors and generators need to be suspended until the environment has properly configured the DUT.

Further, if a testcase needs to inject directed stimulus, it must be able to suspend a transactor or generator for the entire duration of the simulation. If that transactor or generator has already had the opportunity to generate stimulus, it might be impossible to write the required directed testcase.

You might implement transactors as successive derived classes all based on the `vmm_xactor` class. Each inheritance layer might include relevant autonomous threads started in their extension of their respective `vmm_xactor::main()` task.

The execution of the implementation of this task in all intermediate extensions of the `vmm_xactor` base class is necessary for the proper operation of the transactor and control methods.

*Example 4-4 Extension of the vmm_xactor::main() Task*

```
task mii_mac_layer::main();
   fork
      super.main();
   join_none
   ...
endtask: main
```

These methods are virtual to enable the addition of functionality specific to the implementation of a transactor or for you to execute a protocol when you start, stop or reset a transactor.

Note: You should implement transactor methods and invoke its underlying super method.

The implementation of a virtual method in a base class is overloaded in a derived class is only invoked when implicitly called using the *`super`* prefix. When a transactor extends these methods to perform transactor or protocol-specific operations, they must invoke the implementation of these virtual methods in the base class for proper operation.

*Example 4-5 Extension of Control Method*

```
function void
   mii_mac_layer::reset_xactor(reset_e typ = SOFT_RST);
   super.reset_xactor(typ);
   ...
endfunction: reset_xactor
```

You should specify protocols using a layering concept, each with different levels of abstraction. The transactors implementing these protocols should follow a similar division. You can build the functional layer of the verification environment using sub-layers of relevant transactors. For example, a USB functional layer is composed of USB transaction (host or endpoint) and USB transfer (host controller or device) sub-layers.

Master transactors initiate transactions. Slave transactors respond to transactions. A monitor transactor simply observes the interface in master/slave directions, reports observed data as it flows by and any protocol violation it observes. The verification environment shall be able to control the timing of transactions master transactors initiate.

When modeling slave and monitor transactors, you shall take care so that no data is lost if the transactor is executing user-defined callbacks while a significant event occurs on the upstream interface.

This guideline does not imply that a transactor is dynamically reconfigurable, for example, from master to monitor. Due to the significant differences in behavior between modes, it is acceptable to provide this optional configurability using the `vmm_opts` facility. For details, see “Options & Configurations Service” on page 42.

Master and slave transactors should be used when direct interaction with an interface is required to complete or initiate a transaction. When embedding the DUT into a system, that interface might no longer be controllable. Instead, another block controls it in the system.

A monitor transactor should be available to monitor the transactions that are under the control of the block-level environment. This is required to reuse the block-level functional coverage model or its self-checking structure.

You can use notifications by the verification environment to synchronize with the occurrence of a significant event in a transactor or a protocol interface.

In case transaction should be conveyed along with a notification, transactors should post it to a TLM analysis port. For details, see Chapter 5, "Communication".

Even though you implement designs using the same interface protocols, there might be differences in how the protocol is physically implemented by different designs.

Optional elements of the protocol such as bus width, the number of outstanding transactions, clock frequency or the presence of optional side-band signals shall be configurable.

You can specify the configuration of a transactor using a configuration descriptor. All the properties in the configuration descriptor should have the *rand* attribute, however, to allow the generation of random configurations, both to verify the transactor itself under different conditions and to make it usable as a component of the testcase configuration descriptor.

*Example 4-6 MII Transactor Configuration Descriptor*

```
class mii_cfg;
   rand bit is_100Mb;
   rand bit full_duplex;
endclass: mii_cfg
```

Your environment must configure a transactor before using it. The best way to ensure that it configures the transactor is to provide the configuration descriptor as a factory. The transactor might choose to keep a reference to the original configuration descriptor instance or make a copy of it.

In the explicit phasing execution model, the transactors are entirely controlled by explicit method calls from the environment that instantiates it. You should implement this environment by extending `vmm_env`.

The following examples demonstrate how the explicit phase methods are user-extended and called:

*Example 4-7 Modeling Transactor*

```
class my_vip extends vmm_xactor;
   `vmm_typename(my_vip)

  function new(string name = "", vmm_object parent = null);
      super.new("vip", name);
      super.set_parent_object(parent);
   endfunction

   virtual function void start_xactor();
      super.start_xactor();
      `vmm_note(log, "Starting...");
   endfunction

   virtual function void stop_xactor();
      super.stop_xactor();
      `vmm_note(log, "Stopping...");
   endfunction

   `vmm_class_factory(my_vip)
endclass
```

### *Example 4-8 Creation of Explicitly Phased Sub-Environment*

```
class my_subenv extends vmm_subenv;
   `vmm_typename(my_subenv)

   my_vip vip1;
   my_vip vip2;

  function new(string name = "", vmm_object parent = null);
      super.new("vip", name, null);
      super.set_parent_object(parent);
      this.vip1 = new(this, "vip1");
      this.vip2 = new(this, "vip2");
   endfunction

   virtual task start();
      super.configured();
      super.start();
      this.vip1.start_xactor();
      this.vip2.start_xactor();
      `vmm_note(log, "Started...");
   endtask

   virtual task stop();
      super.stop();
      this.vip1.stop_xactor();
      this.vip2.stop_xactor();
      `vmm_note(log, "Stopped...");
   endtask
endclass
```

### *Example 4-9 Creation of Explicitly Phased Environment*

```
class my_env extends vmm_env;
   `vmm_typename(my_env)

   my_subenv subenv1;
   my_subenv subenv2;

   function new();
      super.new("env");
   endfunction
```

```
   virtual function void build();
      super.build();
      this.subenv1 = new("subenv1", this);
      this.subenv2 = new("subenv2", this);
   endfunction

   virtual task start();
      super.start();
      `vmm_note(log, "Started...");
      this.subenv1.start();
      this.subenv2.start();
   endtask

   virtual task wait_for_end();
      super.wait_for_end();
      `vmm_note(log, "Running...");
      #100;
   endtask

   virtual task stop();
      super.stop();
      `vmm_note(log, "Stopped...");
      this.subenv1.stop();
      this.subenv2.stop();
   endtask
endclass
```

### *Example 4-10 Creation of Explicitly Phased Test*

```
`vmm_test_begin(test, my_env, "Test")
   env.run();
`vmm_test_end(test)
```

### *Example 4-11 Top Program*

```
program top;

initial
begin
   my_env env = new;
```

```
   vmm_test_registry::run(env);
end
endprogram
```

## Implicit Phasing

In the implicit phasing execution model, transactors are self-controlled through built-in phasing mechanism. The environment automatically calls the phase specific methods in a top down, bottom up and forked fashion.

Implicit phasing works only with transactors that you base on the `vmm_group` or `vmm_xactor` class. The two use models are,

- If you want to call your transactor phases from the environment, you should instantiate your `vmm_xactor(s)` in `vmm_env or vmm_subenv`.
- If you want to have the environment implicitly calling transactor phases, you should instantiate your `vmm_xactor(s)` in `vmm_group`.

Implicit phasing works only with classes that you base on the `vmm_group` class.

Note: The recommended way of modeling transactor is to extend `vmm_xactor` as it provides a general purpose phasing control. The `vmm_group` class defines several virtual methods that are implicitly invoked during different simulation phases.

*Table 4-2 Predefined Phase and vmm_group Methods*

| Phase | Method | Invocation Order |
|---|---|---|
| RTL config | function rtl_config_ph() | Top down |
| gen_config | function gen_config_ph() | Root objects only |
| build | function build_ph() | Top down |
| configure | function configure_ph() | Bottom up |
| connect | function connect_ph() | Top down |
| configure_test | configure_test_ph | Bottom up |
| start of sim | function start_of_sim_ph() | Top down |
| reset | task reset_ph() | Forked |
| training | task training_ph() | Forked |
| config_dut | task config_dut_ph() | Forked |
| start | task start_ph() | Forked |
| start of test | function start_of_test_ph() | Top down |
| run | task run_ph() | Forked |
| shutdown | task shutdown_ph() | Forked |
| cleanup | task cleanup_ph() | Forked |
| report | function report_ph() | Top down |
| final | function final_ph() | Top down |

You can override any of these methods to implement the required functionality for a particular testbench component for corresponding simulation phase. When overriding a phase method, it is usually recommended that the implementation of the phase method in the base class be executed by calling it through the `super` base class reference.

The following example demonstrates how the implicit phase methods are user-extended then automatically called by the implicit phasing mechanism:

*Example 4-12 Creation of Implicitly Phased Sub-Environment*

```
class my_subenv extends vmm_group;
   `vmm_typename(my_subenv)

   my_vip vip1;
   my_vip vip2;

  function new(string name = "", vmm_object parent = null);
      super.new("vip", name, null);
      super.set_parent_object(parent);
   endfunction

   virtual function void build_ph();
      super.build_ph();
      this.vip1 = new("vip1", this);
      this.vip2 = new("vip2", this);
   endfunction

   virtual task start_ph();
      super.start_ph();
      `vmm_note(log, "Started...");
   endtask
 endclass
```

*Example 4-13 Creation of Implicitly Phased Environment*

```
class my_env extends vmm_group;
   `vmm_typename(my_env)

   my_subenv subenv1;
   my_subenv subenv2;

   function new();
      super.new("env");
   endfunction
```

```
   virtual function void build_ph();
      super.build_ph();
      this.subenv1 = new("subenv1", this);
      this.subenv2 = new("subenv2", this);
   endfunction

   virtual task start_ph();
      super.start_ph();
      `vmm_note(log, "Started...");
   endtask

   virtual task run_ph();
      super.run_ph();
      `vmm_note(log, "Running...");
      #100;
   endtask

   virtual task shutdown_ph();
      super.shutdown_ph();
      `vmm_note(log, "Stopped...");
   endtask
endclass
```

*Example 4-14 Creation of Implicitly Phased Test*

```
class test extends vmm_test;
   function new();
      super.new("Test");
   endfunction

   virtual task start_ph();
      super.start_ph();
      `vmm_note(log, "Started...");
   endtask

   virtual task shutdown_ph();
      super.shutdown_ph();
      `vmm_note(log, "Stopped...");
   endtask
endclass
```

Function phases are invoked in a bottom-up or top-down fashion. However, the order in which functions are executed between two sibling units is not specified.

Tasks phases are forked off to execute concurrently. The order in which the various phase tasks are executed is not specified.

When implementing a transactor or environment, you should avoid relying on a specific order with other components that could be found in the same parent environment. However, such dependencies might not always be avoidable.

## Threads and Processes Versus Phases

It is important to differentiate between execution threads and simulation phases. An execution thread is usually a daemon (For example, a forever loop) that waits for some condition to occur and then performs some task.

For example, a simple master transactor has a thread that waits for a transaction description to arrive on its input transaction-level interface and then executes the transaction you describe.

A phase executes in a finite slice of simulation time to perform a specific functionality, for example, to configure the DUT.

Think of simulation phases as months in a calendar and an execution thread as the behavior of an organism. An organism is born during a specific month, exhibits a specific behavior throughout a certain number of months, goes to sleep, awakens and then dies.

Different organisms are born at different times, exhibit different behaviors, go to sleep and awaken at different times, and might or might not die.

Similarly, transactors are started during specific phases (not necessarily the start phase), run during a certain number of phases. Your environment can suspend their execution thread and resume it, and might stop it during another phase (not necessarily the stop or shutdown phase).

The `vmm_xactor` class is the base class for transactors. It provides thread management utilities (`start`, `stop`, `reset_xactor`, `wait_if_stopped`) that are not present in the other base classes. The `vmm_xactor` offers both thread management and phase methods. It is important to understand to properly model transactors and how you model different behaviors at different phases.

The simplest form for a transactor is one whose behavior does not change between simulation phases. If you instantiate this transactor in an implicitly phased environment, then it gets started by default, for example, you call its `vmm_xactor::start_ph()` method.

```
class vip extends vmm_xactor;
  `vmm_typename(vip)

  virtual task main();
    super.main();
    forever begin
      // Transactor logic meant to run until stopped…
      this.wait_if_stopped();
      ...
    end
  endtask
endclass
```

In the above example, the thread management mechanism `vmm_xactor` base class provides calls the `main()` task. The task is forked off and it continues execution even after the run or shutdown phases complete. You can forcibly abort the execution thread using the `vmm_xactor::reset_xactor()` method.

You should use this method with care as forcibly terminating the execution thread of a transactor might cause an error in the protocol it is executing.

You might stop the execution thread by calling the `vmm_xactor::stop_xactor()` method.

The execution thread will stop at execution points you define by calling the `vmm_xactor::wait_if_stopped()` or `vmm_xactor::wait_if_stopped_or_empty()` methods.

This allows the transaction to stop only when (and if) it is possible; where the protocol is run without causing protocol-level errors.

If a transactor must exhibit different behaviors during different simulation phases, the execution thread might query the current phase you are executing.

```
class my_vip extends vmm_xactor;
   `vmm_typename(my_vip)

  function new(vmm_object parent = null, string name = "");
     super.new("vip", name);
     super.set_parent_object(parent);
   endfunction

   virtual task main();
     vmm_timeline tl = this.get_timeline();
     super.main();
     forever begin
        if (tl.get_current_phase_name() == "config_dut")
```

```
            begin
              `vmm_trace(log, "Config transaction...");
               ...
            end
         else begin
            `vmm_trace(log, "Normal transactions...");
            ...
         end
      end
   endtask
endclass
```

The various phase methods in the transactor might set state variables to different values that affects the execution thread running independently.

```
class my_vip extends vmm_xactor;
   `vmm_typename(my_vip)

  function new(vmm_object parent = null, string name = "");
      super.new("vip", name);
      super.set_parent_object(parent);
   endfunction

   local bit in_config = 0;

   virtual task main();
      super.main();
      forever begin
         if (this.in_config) begin
            `vmm_trace(log, "Config transaction...");
            ...
         end
         else begin
            `vmm_trace(log, "Normal transactions...");
            ...
         end
      end
   endtask
```

```
   virtual task config_dut_ph();
      super.config_dut_ph();
      this.in_config = 1;
   endtask

   virtual task start_ph();
      super.start_ph();
      this.in_config = 0;
   endtask
endclass
```

Though it is possible to modify the behavior of a transactor based on the current simulation phase, it is preferable to avoid it. You define the purpose of simulation phase by the testbench, based on the DUT and test requirements.

To be reusable, a transactor should not enforce specific behaviors at specific phases. In both examples above, it is only possible to execute configuration transactions during the "config_dut" phase.

However, what if you require the execution of such a transaction at another time (for example, to test dynamic reconfiguration or that whether they are properly ignored during normal operations)? It is best to let you decide what behavior you require of a transactor during a particular phase.

## Physical-Level Interfaces

Command-level transactors and bus-functional models are components of the command layer. They translate transaction requests from the higher layers of the verification environment to physical-level signals of the DUT.

In the opposite direction, they monitor the physical signals from the DUT or between two DUT modules. They also notify the higher layers of the verification environment of various transactions the DUT initiates.

The physical-level interface of command-layer transactors must interact with the signal-layer construct. As such, they must follow the guidelines outlined in section “Signal Layer” on page 6.

This specification lets each instance of a transactor to connect to a specific interface instance without hard-coding a signal naming or interfacing mechanism.

The signal layer creates the necessary *`interface`* instances in the top-level module. You can specify the appropriate *`interface`* instance when constructing a transactor, to connect that transactor to that *`interface`* instance.

*Example 4-15 Virtual Interface Connection in Connect Phase Through Encapsulation (Recommended)*

```
// Create interface with appropriate signals and
// connect them to the DUT signals
// at the top module which instantiates DUT.
interface cpu_if (input bit clk);
  wire        busRdWr_N;
  wire        adxStrb;
  wire [31:0]   busAddr;
  wire [ 7:0]   busData;

  clocking cb @(posedge clk);
    output busAddr;
    inout  busData;
    output adxStrb;
    output busRdWr_N;
  endclocking

  modport drvprt(clocking cb);
```

```
endinterface

//Instantiate interface in top level module
// and connect to DUT signals
module test_top();

  //Interface instantiation
  cpu_if cpuif(clk);

  //DUT instantiation
  cntrlr dut(.clk(clk),
             .reset(reset),
             .busAddr(cpuif.busAddr),
             .busData(cpuif.busData),
             .busRdWr_N(cpuif.busRdWr_N),
             .adxStrb(cpuif.adxStrb));
endmodule

// Create a vmm_object wrapper which gets virtual
// interface handle through the constructor.
// Instantiate the object with the actual interface
// instantiation and allocate the vmm_object
// instance to appropriate transactor through
vmm_opts::set_object() method.

class cpuport extends vmm_object;
   virtual cpu_if.drvprt iport;

  function new(string name,virtual cpu_if.drvprt iport);
     super.new(null, name);
     this.iport = iport;
   endfunction

endclass

program cntrlr_tb;
   cpuport cpu_port;  //Interface wrapper
   cntrlr_env    env;
   initial begin
      env = new("env");
   //Instantiating with the actual interface
   // instance path
```

```
      cpu_port = new("cpu_port",test_top.cpuif);


vmm_opts::set_object("CPU:CPUDrv:cpu_port",cpu_port,
env); //Sending the wrapper to driver
  end
endprogram

// use vmm_opts::get_object_obj() method
// (or `vmm_unit_configure_obj macro)
// to get the object wrapper instance and hence the
// virtual interface handle
// in the vmm_xactor::connect_ph(). Since it is a dynamic
// allocation, it is recommended
// to have a null object check on the virtual interface
// instance before using it.
class cpu_driver extends vmm_xactor;

  virtual cpu_if.drvprt  iport;
  cpuport cpu_port_obj;

  virtual function void connect_ph();
    bit is_set;
    if ($cast(cpu_port_obj,
vmm_opts::get_object_obj(is_set,this,"cpu_port")))
    begin
      if (cpu_port_obj != null)
       this.iport = cpu_port_obj.iport;
      else
       `vmm_fatal(log, "Virtual port wrapper not
initialized");
   end
  endfunction

endclass
```

The *clocking* block separates timing and synchronization of synchronous signals from the reference signal. It defines the timing and sampling relationships between synchronous data and clock signals.

If a transactor waits for the next edge of the clock by using an *@(posedge...)* statement, it might wait for the wrong active edge or the wrong clock signal compared to the one specified in the *clocking* block. It might sample the wrong value of the synchronous signals. To wait for the next cycle of synchronous signals, use the *@* operator with a clocking block reference.

*Example 4-16 Using @ Operator to Synchronize BFMs*

```
task mii_mac_layer::tx_driver();
   ...
   @this.sigs.mtx;
   this.sigs.mtx.txd <= nibble;
   ...
endtask: tx_driver

task mii_mac_layer::rx_monitor();
   ...
   @(this.sigs.mrx);
   if (this.sigs.mrx.rx_dv !== 1'b1) break;
   a_byte[7:4] = this.sigs.mrx.rxd;
   ...
endtask: rx_monitor
```

## Transactor Callbacks

The behavior of a transactor shall be controllable as the verification environment and individual testcases require without modifications of the transactor itself.

These requirements are often unpredictable when you first write the transactor. By allowing the execution of arbitrary user-defined code in callback methods, you can adapt the transactors to the needs of an environment or a testcase. For example, you can use callback methods to monitor the data flowing through a transactor to check for correctness, inject errors or collect functional coverage metrics.

The actual set of callback methods that you must provide by a transactor is protocol-dependent. Subsequent guidelines will help design a suitable set in most cases. You should provide additional callback methods required by the protocol or the transactor implementation.

Whether it is a transaction descriptor or sampling a byte on a physical interface, the new input data should be reported to you through a post-reception callback method. It should be recorded in or checked against a scoreboard and modified to inject an error or collect functional coverage metrics.

Whether it is a transaction descriptor or driving a byte on a physical interface, the new output data should be reported to you through a pre-transmission callback method. It should be recorded in or checked against a scoreboard and modified to inject an error or collect functional coverage metrics.

Whenever a transaction requires locally generated additional information, the additional information should be reported to you through a post-generation callback method. It should be recorded in or checked against a scoreboard and modified to inject an error or collect functional coverage. You should provide a reference to the original transaction to convey context information.

For example, a transactor prepending a packet with a preamble should call a callback method with the generated preamble data before starting the transmission process.

Whenever a transactor makes a choice among several alternatives, the choice and available alternatives should be reported to you through a post-decision callback method. It should be recorded in or checked against a scoreboard and modified to select another alternative or collect functional coverage.

All information relevant to the context of the decision-candidates, rules and alternatives-should be provided to you along with the default decision via the callback method.

For example, a transactor selecting traffic from different priority queues should call a callback method after selecting a queue based on the current priority selection algorithm; however before pulling the next item from the selected queue. You can then modify the selection.

This declaration creates a façade for all available callback methods for a particular transactor. You require the common base class to be able to register the callback extension instances using the predefined methods and properties in the `vmm_xactor` class.

If the transactor implementation or protocol can support delays in the execution of a callback, you should declare it as a task. You should declare callbacks that must be non-blocking as a function.

Restricting callback functions to `void` functions avoids difficulties with handling a return value from a function when you register multiple callback extensions and cascade in a transactor. It should return by modifying an instance referred to by an argument or a scalar argument passed by reference to various status informations returned from a callback method (such as a flag to indicate whether to drop the transaction).

You cannot modify callback arguments as it would break the implementation of the transactor. You should not modify others to avoid creating inconsistencies within the transaction being executed or observed.

You can modify arguments without the `const` attribute, however to inject errors.

This inclusion allows registration of one extension of the callback methods with more than one transactor instance and identifies which transactor has invoked the callback method.

Callback should be registered in the *vmm_xactor* base class. However, calling the registered callback extensions is the responsibility of the transactor extended from the base class.

To remove the transactor implementation from the details of callback registrations and to ensure that you call them in the proper registration sequence, you use this macro to invoke the callbacks.

*Example 4-17 Transactor Callback Usage*

```
// Create a callback class with empty virtual methods
// Each virtual method represents an important stage
// of transactor.
// The arguments of the virtual methods should contain
// necessary information that can be
// shared with the subscribers.

class cpu_driver_callbacks extends vmm_xactor_callbacks;
  virtual task pre_trans  (cpu_driver driver, cpu_trans
tr, ref bit drop);
   endtask
  virtual task post_trans  (cpu_driver driver, cpu_trans
tr);
   endtask
endclass

// At every important stage in the transactor,
// call the corresponding method
//  through `vmm_callback macro.

class cpu_driver extends vmm_xactor;

   virtual protected task main();
          super.main();
          ……
    `vmm_callback(cpu_driver_callbacks, pre_trans(this,
```

```
tr, drop));
       if (tr.kind == cpu_trans::WRITE) begin
         write_op(tr);
       end
       if (tr.kind == cpu_trans::READ) begin
         read_op(tr);
       end
       `vmm_callback(cpu_driver_callbacks,
post_trans(this, tr));
   endtask

endclass

// A subscriber extend the callback class, fill
// the necessary empty virtual methods.

class cpu_sb_callback extends cpu_driver_callbacks;
   cntrlr_scoreboard sb;

   function new(cntrlr_scoreboard sb);
       this.sb = sb;
   endfunction

  virtual task pre_trans(cpu_driver drv, cpu_trans tr,ref
bit drop);
     sb.cpu_trans_started(tr);
  endtask

  virtual task post_trans(cpu_driver drv, cpu_trans tr);
     sb.cpu_trans_ended(tr);
  endtask

endclass

// Register the subscriber callback class using method
// vmm_xactor::append_callback.
// Then every time transactor hits the defined important
// stages, subscriber methods
// will be called. Note that any number of subscribers
// with their own definition of virtual
// methods can get registered to a transactor.
```

```
class cntrlr_env extends vmm_group;
   cpu_driver drv;

    virtual function void connect_ph();
           cpu_sb_callback cpu_sb_cbk = new(sb);
           cpu_cov_callback cpu_cov_cbk = new(cov);
           drv.append_callback(cpu_sb_cbk);
           drv.append_callback(cpu_cov_cbk);
    endfunction

endclass
```

# Advanced Usage

## User-defined vmm_xactor Member Default Implementation

For the `vmm_xactor` class, you accomplish this by using the `'vmm_xactor_member_user_defined()` macro and implementing a function named `"do_membername()`.

You implement this function using the following pattern:

```
function bit do_name(vmm_xactor::do_what_e do_what,
                     vmm_xactor::reset_e   rst_typ);
   do_name = 1; // Success, abort by returning 0

   case (do_what)
      DO_PRINT: begin
         // Add to the 'this.__vmm_image' variable,
         // using 'this.__vmm_prefix'
      end
      DO_START: begin
         // vmm_xactor::start_xactor() operations.
      end
      DO_START: begin
         // vmm_xactor::stop_xactor() operations.
      end
```

```
      DO_RESET: begin
         // vmm_xactor::reset_xactor() operations.
      end
   endcase
endfunction
```

Note: You must provide a default implementation for all possible operations (print, consensus registration, start and stop). It is not possible to execute the default implementation that you would otherwise provide by the other type-specific shorthand macros. However, it is acceptable to leave the implementation for an operation empty if you are not going to use it or it has no functional effect.

## User-Defined Implicit Phases

Adding user-defined phases in an implicitly phased environment is a simple task of adding additional virtual methods that you must call in the appropriate sequence.

In an implicitly phased environment, user-defined phases you might insert between the pre-defined phases by any component in the environment. You might insert a new phase in a timeline or aliased to an existing phase to execute concurrently.

It is important to note that any phase that executes before the `"build"` phase executes on the root objects only, because the object hierarchy has not been built yet.

You should add user-defined phases to the parent timeline of the component that creates it. This way, should the component be encapsulated in a sub-timeline, its user-defined phase will be added

to the encapsulating sub-timeline. By this you allow potentially conflicting user-defined phase definitions to be kept in separate timelines.

A user-defined phase executes a *void* function or a task in various user-defined class extension of the `vmm_object` base class. For example, you could add a phase to call the `vip::delay_ph()` in all instances of the `vip` class.

```
class vip extends vmm_xactor;
   `vmm_typename(vip)
   ...
   task delay_ph();
      #(this.delay);
   endtask
endclass
```

First, you need to implement a user defined phase wrapper extending from `vmm_fork_task_phase_def`. This base class is chosen because the phase method is a task. If the phase method were a function, the wrapper would have implemented using an extension of `vmm_topdown_function_phase_def` or `vmm_bottomup_function_phase_def`.

```
class vip_delay_ph_def extends vmm_fork_task_phase_def
#(vip);
   `vmm_typename(vip_delay_ph_def)

   virtual task do_task_phase(vip obj);
      if(obj.is_unit_enabled()) obj.delay_ph();
   endtask
endclass
```

You then add the new user defined phase definition to the parent timeline at an appropriate point. You do this in the `build` phase as shown in the following example:

```
class vip extends vmm_xactor;
  ...
  virtual function void build_ph();
     vmm_timeline tl = this.get_timeline();
     vip_delay_ph_def ph = new;

     //schedule vip_delay phase execution before reset
     tl.insert_phase("vip_delay", "reset", ph);
  endfunction
  ...
endclass: vip
```

Inserting phases in the environment is done in exactly the same way, since an environment is also a `vmm_group`.

Inserting phases in a test is identical, however, with a small difference. The `vmm_test` class derives from `vmm_timeline`, therefore, you can insert user-defined phase directly by calling `this.insert_phase` directly.

The following example shows the insertion of `delay_ph()` in a test, before the reset phase.

```
class test1 extends vmm_test;
   `vmm_typename(test1)
   ...
   virtual task delay_ph();
    #(env_cfg.test_delay);
   endtask

  virtual function void build();
     test_delay_ph_def ph = new;

     //schedule test_delay phase execution before reset
     this.insert_phase("test_delay", "reset", ph);
  endfunction
endclass
```

During implicit phasing, when you encounter a `vmm_timeline` object, you execute its phases up to the currently executing phase (with the same name, if present) in the higher-level timeline. This allows sub-timelines to create phases that do not exist in the top-level phase.

## Skipping an Implicit Phase

You can use the `vmm_null_phase_def` class used to override a predefined or existing phase to skip its implementation for a specific `vmm_group` instance.

The following example shows how you can skip the pre-defined `"start"` phase in the vip1 transactor present in the environment in a testcase, to prevent it from starting automatically.

```
class test1 extends vmm_test;
   `vmm_typename(test1)
   my_env env;

   virtual function void build();
      vmm_null_phase_def nullph = new;

      env.vip1.override_phase("start", nullph);
   endfunction
endclass
```

## Disabling an Implicit Component

For a test specific objective or to debug part of the code, you might want to disable one or more unit instances. Similarly, when composing system-level environments from block-level

environments, you might find it necessary to disable some block-level testbench components because their function is no longer relevant within the system-level context.

A disabled unit instance (and all of its children objects) is no longer considered by the timeline to which it belongs. It is no longer part of the implicit phasing mechanism. You can disable a unit instance as follows:

```
class top extends vmm_group;
   `vmm_typename(top_unit)
   ahb_driver drv0,drv1;
   virtual function void build_ph();
      drv0 = new(”drv0”, this);
      drv1 = new(”drv1”, this);
   endfunction
endclass

//single driver test
class my_test1 extends vmm_test;
   `vmm_typename(my_test1)

   virtual function void configure_ph();
      // Disable drv1
      top env = vmm_object::find_object_by_name("top");
      env.drv1.disable_unit();
  endfunction
endclass
```

## Synchronizing on Implicit Phase Execution

Each phase has associated events and status flags which are available to synchronize with the execution of a particular phase during simulation.

you might use the `vmm_phase::is_done()` method to check the execution status of any phase. For a particular timeline, calling `is_done()` on that phase will return the number of times the phase has executed completely.

```
begin
   vmm_timeline top = vmm_simulation::get_top_timeline();
   vmm_phase    ph  = top.get_phase("connect");
   wait(ph.is_done() == 1);
end
```

The `vmm_phase::is_running()` method checks the status for any task phase. This is not meaningful for any function phase, since the phase executes in zero time and the result of the `vmm_phase::is_running()` method will always be 0, unless `vmm_phase::is_running()` is called within that function phase.

The `vmm_phase::completed` and `vmm_phase::started` events get triggered when the execution of a phase completes and starts, respectively.

```
begin
    vmm_timeline top = vmm_simulation::get_top_timeline();
    vmm_phase ph = top.get_phase("reset");
    fork
        begin
           @(ph.started);
           `vmm_note(log, " reset phase is running");
        end
        begin
          @(ph.completed);
           `vmm_note(log, " reset phase is completed");
        end
    join
 end
```

## Breakpoints on Implicit Phasing

The `+vmm_break_on_*` command-line options are available to interrupt the execution flow at specific phases, either globally or for a specific timeline. For example, the `+vmm_break_on_phase+reset` command-line option causes the phasing to be interrupted at the start of the reset phase. These options may also be specified from within the code using:

```
vmm_opts::set_string("break_on_phase", "reset");
```

By default, `$stop` is called when the phasing is interrupted. However, if callbacks are registered with the timeline, the registered `vmm_timeline_callback::break_on_phase()` method(s) will be called instead.

If you do not specify the instance name with the command-line option, root timeline is interrupted before the specified phase (if present in root timeline).

To interrupt specific timeline instances, specify the hierarchical name of the timeline to be interrupted using:

```
vmm_opts::set_string("break_on_timeline", timeline_name);
```

Here are additional details on the different options available to you for debugging phases and timelines.

`+vmm_break_on_phase`

> Specifies "+" separated list of phases on which to break. If you have provided this, and either haven't passed `+vmm_break_on_timeline` or have provided invalid name for timeline, you break out on root level timelines (pre/top/post).

`+vmm_break_on_timeline`

Specifies "+" separated list of timelines on which to break. If you have specified this option and, either haven't passed `+vmm_break_on_phase` or have provided invalid name for phase, it is ignored.

Note:

Instead of specifying the timeline name, you can also specify the pattern of name.

`+vmm_list_phases`

Lists down available phases in simulation at the end of the pre-test timeline. This comes into effect when `vmm_simulation::run_tests()` is used to run the simulation.

`+vmm_list_timeline`

Lists down available timelines in simulation at the end of the pre-test timeline. This comes into effect when `vmm_simulation::run_tests()` is used to run the simulation.

Note:

- If you have specified to break on a particular phase with a particular timeline and that timeline is created in build phase (or later), then we will break twice, once on pre_test and then on the actual timeline.
- If you have specified a valid phase name and a valid timeline name, however the specified timeline doesn't contains that particular phase, we don't break on anything.

**Examples**

To list all timelines in simulation:

```
./simv +vmm_list_timeline
```

To list all phases in each of the timelines in simulation:

```
./simv +vmm_list_phases
```

To break on root timeline in phase X:

```
./simv +vmm_break_on_phase=X
```

To break on timeline A in phase X:

```
./simv +vmm_break_on_phase=X +vmm_break_on_timeline=A
```

To break on timeline A & B in phase X:

```
./simv +vmm_break_on_phase=X +vmm_break_on_timeline=A+B
```

To break on root timeline in phase X and Y:

```
./simv +vmm_break_on_phase=X+Y
```

To break on timeline A & B in phases X&Y:

```
./simv +vmm_break_on_phase=X+Y +vmm_break_on_timeline=A+B
```

## Concatenation of Tests

In case of multiple tests `top_test` timeline is reset to the phase identified as start phase for the test you need to execute. The test can specify itself concatenable and specify the starting phase by using `vmm_test_concatenate()` macro.

```
class test_concatenate1 extends vmm_test;
  //Macro to indicate the rollback phase in
```

```
  //case of test concatenation
  `vmm_test_concatenate(configure_test)

  function new(string name);
     super.new(name);
  endfunction

  virtual function void configure_test_ph();
     vmm_opts::set_int("%*:num_scenarios", 20);
     cpu_rand_scenario::override_with_new(
        "@%*:CPU:rand_scn",
          cpu_write_read_same_addr_scenario::this_type(),
        log, `__FILE__, `__LINE__);
  endfunction

endclass

class test_concatenate2 extends vmm_test;
  //Macro to indicate the rollback phase in case of test
  //concatenation
  `vmm_test_concatenate(configure_test)

  function new(string name);
     super.new(name);
  endfunction

  virtual function void configure_test_ph();
     vmm_opts::set_int("%*:num_scenarios", 20);
     cpu_rand_scenario::override_with_new(
        "@%*:CPU:rand_scn",
        cpu_write_scenario::this_type(),
        log, `__FILE__, `__LINE__);
  endfunction

endclass

//Command line arguments to run the example
./simv +vmm_test=test_concatenate1+test_concatenate2
```

## Explicitly Phasing Timelines

You have several options to explicitly control the step-by-step progress of implicit phase execution in a timeline object.

You might use the `vmm_timeline::run_phase()` and `vmm_timeline::run_function_phase()` methods to run the timeline up to and including the specified phase.

Note: All phases to be executed must be function phases.

```
class my_subenv extends vmm_timeline;
   my_vip vip;

   virtual function void build();
      super.build();
      this.vip = new(this,"vip");
   endfunction
   ...
endclass

class my_env extends vmm_env;
   vmm_timeline tl;

   virtual function void build();
      super.build();
      this.tl = new("tl", this);
      this.tl.run_function_phase("build");
   endfunction

   virtual task reset_dut();
      super.reset();
      this.tl.run_phase("reset");
   endtask

   virtual task config_dut();
      super.config_dut();
      this.tl.run_phase("config_dut");
   endtask
   ...
```

```
endclass
```

The `vmm_timeline::reset_to_phase()` method may be used to rollback the timeline to the specified phase.

# 5

# Communication

This chapter contains the following sections:


- “Overview”
- “Channel”
- “Completion Using Notification (vmm_notify)”
- “Transport Interfaces in OSCI TLM2.0”
- “Broadcasting Using TLM2.0”
- “Interoperability Between vmm_channel and TLM2.0”
- “Advanced Usage”

## Overview

This section applies to the transaction-level interfaces connecting independent transactors.

Transaction-level interfaces are mechanisms to exchange transactions between two independent blocks such as between two transactors or a directed testcase and a transactor.

In command-layer transactors such as drivers and monitors, the transaction-level interface allows the higher layers of the verification environment to stimulate the DUT by specifying which transactions should be executed. Also, the higher layers can be notified of transactions that have been observed on a DUT interface.

VMM supports multiple ways of passing transactions between transactors. The supported interfaces are:

- Channel
- TLM Blocking transport
- TLM Non-Blocking transport
- TLM Analysis port
- Callback

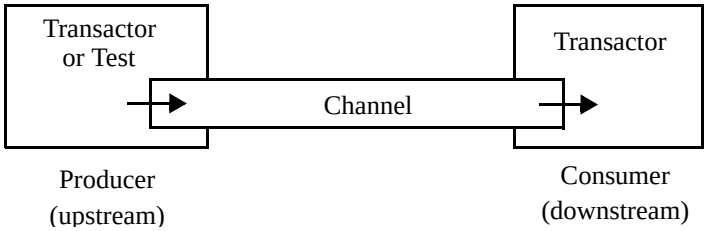
## Channel

A connection can be established between two transactors or a testcase and a transactor by having each endpoint, the producer and the consumer, refer to the same conduit. This is shown in Figure 5-

1. You can make the connection by instantiating the endpoints in any order to allow the bottom-up or top-down building of verification environments.

The conduit allows a transactor, whether upstream or downstream to connect to any other transactor with a compatible conduit. This occurs without any source code modification requirement or knowledge of the other endpoint.

*Figure 5-1 Transaction Interface Channel*




Traditionally, transaction-level interfaces are implemented using procedure calls in the transactors themselves. However, invoking a procedure in a transactor instance requires a reference to that transactor in the first place. This limitation requires that verification environments are built bottom-up, with the higher layers having a reference to the lower-level transactor instances methods in them.

This structure creates some difficulties. You cannot build a verification environment on top of the physical layer that you can then retarget, without modifications to a different physical-layer implementation.

By encapsulating the transaction exchange mechanism into a conduit, you consider the transactors as endpoints to the conduit that you can replace easily, for knowledge by or of the other endpoint is no longer required.

VMM uses the channel class as the conduit between the endpoints. Each connection between two endpoints requires a transport-interface instance.

## Channel Declaration (vmm_channel_typed)

It is also possible to define a channel by using the parameterized class `vmm_channel_typed`. This can be very useful when embedding the channel in another parameterized class. Here, the transaction is passed from the parent class to the channel.

Alternatively, macros can be used in the same way. However, using a parameterized class is easier to debug than macros.

Note: `vmm_channel_typed` is not tied to `vmm_channel` and can transport any kind of object.

Example 5-1 shows how to declare the *`eth_frame_channel`* using *`vmm_channel_typed`* class.

*Example 5-1 Defining a Transaction Channel Using vmm_channel_typed*

```
class eth_frame extends vmm_data;
   rand bit [47:0] da;
   rand bit [47:0] sa;
   rand bit [15:0] len_typ;
   rand bit [7:0]  data [];
   rand bit [31:0] fcs;

   `vmm_data_member_begin(eth_frame)
      `vmm_data_member_scalar(da, DO_ALL)
      `vmm_data_member_scalar(sa, DO_ALL)
      `vmm_data_member_scalar(len_typ, DO_ALL)
      `vmm_data_member_scalar_array(data, DO_ALL)
      `vmm_data_member_scalar(fcs, DO_ALL)
   `vmm_data_member_end(eth_frame)
endclass

typedef vmm_channel_typed #(eth_frame)
```

```
eth_frame_channel;
```

## Channel Declaration (vmm_channel)

The `vmm_channel` is a template class that is defined specifically for the data or transaction descriptor it carries. A channel class is easily defined for each `vmm_data` derivative as the data or transaction descriptor class name with the `"_channel"` suffix.

In Example 5-2, the class `eth_frame_channel` is defined to carry instances of the `eth_frame` transaction.

*Example 5-2 Defining a Transaction Channel Using ‘vmm_channel*

‘vmm_channel(eth_frame)

## Connection of Channels Between Transactors

You cannot use an `interface` as a transaction-level interface because, like a `module`, it is a static construct. It is not possible to create dynamically reconfigurable verification environments using interfaces.

Furthermore, you do not build interfaces on top of the object-oriented framework and cannot derive from one another. It is therefore not possible to provide common functionality through a base `interface` like it is possible through a channel base `class`.

As described in “Implicit Phasing” on page 14, it is possible to model transactors so they are *implicitly* controlled. Here, you do not have to worry about transactor channel connection in the transactor itself as in the environment or sub-environment phases.

*Example 5-3 Declaring and Connecting Channel Instances in Implicitly Phased Environment*

```
class eth_subenv extends vmm_group;
   eth_frame_channel tx_chan;
   eth_frame_channel rx_chan;
   eth_mac mac;
   mii_mac mii;
   ...

   function build_ph();
      tx_chan = new("TxChan", "TxChan0");
      rx_chan = new("RxChan", "RxChan0");
      mac = new(this, "Mac");
      mii = new(this, "Mii");
   endfunction

   function connect_ph();
      mac.pls_tx_chan = tx_chan;
      mac.pls_rx_chan = rx_chan;
      mii.tx_chan = tx_chan;
      mii.rx_chan = rx_chan;
   endfunction
   ...
endclass
```

Note: The channel connection is done differently if you instantiate the transactor in an *explicitly* phased environment or sub-environment. As the `connect` phase is not available in `vmm_env` or `vmm_subenv`, it is recommended to connect channels in the `build()` explicit method.

*Example 5-4 Declaring and Connecting Channel Instances in Explicitly Phased Environment*

```
class eth_subenv extends vmm_subenv;
   eth_frame_channel tx_chan;
   eth_frame_channel rx_chan;
   eth_mac mac;
   mii_mac mii;
   ...
```

```
   function build();
      tx_chan = new("TxChan", "TxChan0");
      rx_chan = new("RxChan", "RxChan0");
      mac = new(this, "Mac");
      mii = new(this, "Mii");
      mac.pls_tx_chan = tx_chan;
      mac.pls_rx_chan = rx_chan;
      mii.tx_chan = tx_chan;
      mii.rx_chan = rx_chan;
   endfunction
   ...
endclass
```

It is recommended not to connect channels using the transactor constructor. This approach is not reusable and it is impossible to replace the channel by a factory.

## Declaring Factory Enabled Channels

You should use the factory service class to ensure that channels used in the environment for transactor communication are replaceable by another channel of the same type.

This is an important aspect of reuse and allows dynamically rebuilding environments and connecting transactors to different channels during the simulation or between tests.

Making a channel factory-enabled is achieved by replacing the channel `new()` constructor with the `factory::create_instance()`.

*Example 5-5 Declaring Factory Enabled Channels*

```
class eth_subenv extends vmm_group;
   ...
```

```
    function build_ph();
       tx_chan = eth_frame_channel::create_instance(
                    this, "TxChan");
       rx_chan = eth_frame_channel::create_instance(
                    this, "RxChan");
    endfunction
    ...
endclass
```

## Overriding Channel Factory

You should use the `override_with_copy()` method to replace a channel by another channel of the same type with a regular expression that matches the transactor. This expression is attached to this channel.

There are many applications where you can use this replacement such as connecting another multi-stream-scenario generator channel to a transactor or connecting a transactor to a different scoreboard, reference model, coverage model, etc.

*Example 5-6 Overriding Channels Using Factory*

```
class my_test extends vmm_test;
   ...
   eth_frame_channel new_tx_chan;

   function start_of_sim();
     new_tx_chan = new("NewTxChan", ...);
     eth_frame_channel::override_with_copy(
                  "@*:eth_subenv", nw_tx_chan, log);
   endfunction
```

## Channel Completion and Response Models

You can provide transaction descriptors to transactors through a channel instance. It is usually important for the higher-layer transactors to know when you complete a transaction by a lower-layer transactor. It is also important for them to know how to respond to a reactive transactor.

Furthermore, it must be possible for a transactor to output status information about the execution of the transaction.

A completion model is used by transactors to indicate the end of a transaction execution. A response model is used by a reactive transactor to request additional data or information required to complete a suitable response to the transaction being reacted to, from the higher layers of a verification environment.

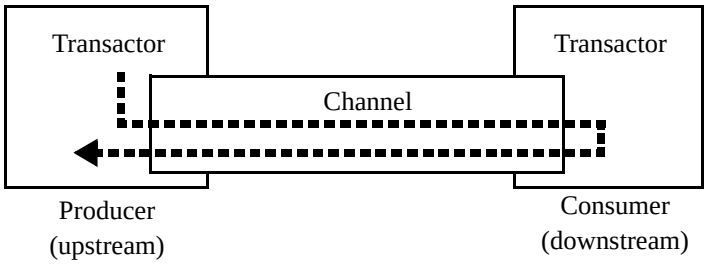
The completion and response models can be modeled using a producer transactor and a consumer transactor. The consumer transactor executes transactions requested by the producer transactor and indicates completion and response information back to the producer transactor.

## Typical Channel Execution Model

Usually transactors execute transactions in the same order as they are submitted. Each transaction is executed only once and it completes in a single execution attempt.

Such transactors use a blocking completion model. As shown in Figure 5-2, the execution thread is blocked from the producer transactor (depicted as a dotted line) while the transaction flows through the channel and the consumer transactor executes it. It remains blocked until the execution of the transaction is completed.

*Figure 5-2 In-Order Atomic Completion Model*




From the producer transactor's perspective, `vmm_channel::put()` method embodies the blocking completion model. When this method returns, the transaction completes. You might add additional completion status information to the transaction descriptor by using the consumer transactor.

*Example 5-7 Upstream of a Blocking Completion Model*

```
class producer extends vmm_xactor;
   ...
   virtual task main();
      ...
      ... begin
         transaction tr;
         ...
         do
            out_chan.put(tr);
         while (tr.status == RETRY);
         ...
      end
      ...
   endtask: main
endclass: producer
```

Note: This channel becomes *blocking* if the channel can only retain one transaction and the attached transactor has not carried out a *vmm_channel::get()* method access. Any other configuration creates a *non-blocking* interface.

To ensure that input channels are "full", and therefore blocking when there is one transaction in the channel, consumer transactors must explicitly reconfigure the input channel instances.

*Example 5-8 Reconfiguring an Input Channel Instance*

```
class consumer extends vmm_xactor;
   transaction_channel in_chan;
   ...
   function void start_of_sim_ph();
     this.in_chan.reconfigure(1);
   endfunction: start_of_sim_ph
   ...
endclass: consumer
```

To ensure the producer does not push a new transaction right after the `vmm_channel::put()` and that this transaction remains unchanged while it's being processed, you should use the `vmm_channel::peek()` or `vmm_channel::activate()` method to obtain the next transaction you execute from the input channel.

*Example 5-9 Peeking Transaction Descriptors*

```
class consumer extends vmm_xactor;
   ...
   virtual task main();
      ...
      forever begin
         transaction tr;
         this.in_chan.peek(tr);
         ...
         this.in_chan.get(tr);
      end
   endtask: main
   ...
endclass: consumer
```

A transaction is removed from a channel by using the *vmm_channel::get()* or *vmm_channel::remove()* method.

If the transaction descriptor has properties that can be used to specify completion status information, these properties may be modified by the *consumer* transactor to provide status information back to the *producer* transactor.

*Example 5-10 Providing Status Information in a Transaction Descriptor*

```
class consumer extends vmm_xactor;
   ...
   virtual task main();
      ...
      forever begin
         transaction tr
         ...
         this.in_chan.start(tr);
         ...
            tr.status = ...;
         ...
         tr.in_chan.complete();
         ...
      end
   endtask: main
endclass: consumer
```

If the transaction descriptor does not have properties that can be used to specify completion status information, the consumer transactor can provide status information back to the producer.

There are many other operating modes that can be supported by vmm_channel. For details, see “Advanced Usage” on page 43.

## Channel Record/Playback

VMM channel provides a facility to record the transactions going through and save them into a file. You can then playback these transactions from the same file. As playback avoids randomization of the transaction/corresponding scenarios, you can improve performance in case of complex transaction/scenario constraints.

Also, generation is not scheduling-dependent and will work with different versions of the simulator and with different simulators. You can use this record/replay mechanism to go through known states at one interface while stressing another interface with random scenarios within the same simulation itself. This guarantees you random stability.

The use model is as follows:

- Model your transaction using shorthand macros so that it records/replays and stores its information in a consistent way
- Record incoming transactions through `vmm_channel::record()` method
- Replay recorded transactions through `vmm_channel::playback()` method

*Example 5-11 Using vmm_channel::record() and vmm_channel::playback()*

```
class my_subenv extends vmm_group;
   typedef enum { NORMAL , RECORD , PLAYBACK } rp_mode;
   string md;
   my_mode mode;
   string filename = "tx_chan.dat";

   eth_frame_channel tx_chan;
   eth_mac mac;
```

```
mii_mac mii;
eth_frame fr;
...

function build_ph();
   tx_chan = eth_frame_channel::create_instance(
                  this,"TxChan");
   mac = new(this, "Mac");
   mii = new(this, "Mii");
endfunction

function configure_ph();
   // Enable run time option to specify the
   // record/playback mode
   // Available with _vmm_opts_mode=MODE
   md = vmm_opts::get_string("MODE", // Switch name
                             "NORMAL" , // Default
                          "Specifies the mode"); // Doc
   case(md)
      "NORMAL" : mode = NORMAL;
      "RECORD" : mode = RECORD;
      "PLAYBACK" : mode = PLAYBACK;
   endcase
endfunction

function connect_ph();
   mii.tx_chan = tx_chan;
   case(mode)
      NORMAL: begin
         mac.pls_tx_chan = tx_chan;
      end;
      RECORD:  begin
         // record all eth_frame to tx_chan.dat
         mac.pls_tx_chan = tx_chan;
         tx_chan.record(filename);
      end;
      PLAYBACK:  begin
         // playback eth_frame from tx_chan.dat
         // Don't connect the mac xactor
         tx_chan.playback(success, filename, fr);
         if(!success)
            `vmm_error(log,
```

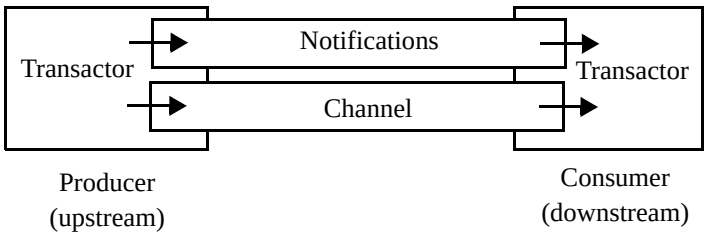
```
                    "Playback mode failed for channel");
            end;
         endcase
      endfunction
      ...
endclass
```

## Completion Using Notification (vmm_notify)

Though the channel offers a rich set of completion models, it can only provide transaction information after-the-fact. A channel transfers a data or transaction descriptor only once a consumer completely receives it.

In some protocols or circumstances, higher-layer transactors require timing-related information as soon as that information is available asynchronously from any transaction completion it is associated with. For example, a MAC layer Ethernet transactor needs to know when the medium is busy so it can defer the transmission of various frames it might have. If the MAC layer Ethernet transactor delays the information until you receive the frame occupying the medium completely, then it becomes stale.

*Figure 5-3 Notification Interface*




## Notification Service Class

Transaction-asynchronous timing information can be exchanged between two transactors via an instance of a notification service interface. One transactor produces notification indications while the other waits for the relevant indications. A parallel channel can be used to transfer any transaction information once it is complete.

The extension of the `vmm_notify` class defines all notifications that can be exchanged between the two transactors.

*Example 5-12 Notification Service Class*

```
class eth_pls_indications extends vmm_notify;
   typedef enum {CARRIER, COLLISION} indications_e;

   function new(vmm_log log);
      super.new(log);
      super.configure(CARRIER, ON_OFF);
      super.configure(COLLISION, ON_OFF);
   endfunction: new
endclass: eth_pls_indications
```

This structure allows the connection between two transactors in an arbitrary order. The first one creates the notification service instance; the second uses the reference to the instance in the first one.

*Example 5-13 Notification Service Class Property*

```
class eth_mac extends vmm_xactor;
   ...
```

```
   eth_pls_indications indications;
     ...
endclass: eth_mac
```

For example, when channels connecting two transactors require that they share a reference to the same notification service instance, it should be possible to specify notification service instances to connect as optional constructor arguments. If none is specified, new instances are internally allocated.

In Example 5-14, the notification service instances are allocated if none is specified via the constructor argument list.

*Example 5-14 Optional Notification Service Instances in Constructor*

```
class eth_mac extends vmm_xactor;
   eth_pls_indications indications;
   ...
   function new(...
                eth_pls_indications indications = null);
      ...
      if (indications == null) indications = new(...);
      this.indications = indications;
      ...
   endfunction: new
   ...
endclass: eth_mac
```

If a transactor holds a copy of the reference to a notification service instance in an internal variable, the notification service instance cannot be substituted with another to modify the output or input of a transactor and dynamically reconfigure the structure of a verification environment.

While it is unavoidable during normal operations, a reset or stopped transactor should release all such internal references to allow the replacement of the notification service instance.

## Notify Observer

VMM Notify Observer simplifies subscription to a notify callback class. It is a parameterized extension of `vmm_notify_callbacks`. Any subscriber (such as, a scoreboard, coverage model, etc.) can get the transaction status whenever you indicate a notification event. Call the `` `vmm_notify_observer `` macro, specifying the observer and its method name.

```
class vmm_notify_observer #(type T, type D = vmm_data)
   extends vmm_notification_callbacks
```

Consider a subscriber such as a scoreboard having a method named `observe_trans()`. Define a `` `vmm_notify_observer `` macro specifying the subscriber name (`scoreboard`) and the method name(`observe_trans`).

```
class scoreboard;
   virtual function void observe_trans(ahb_trans tr);
      ...
   endfunction
endclass
`vmm_notify_observer(scoreboard, observe_trans)
```

You can instantiate the parameterized `vmm_notify_observer` by passing its subscriber handle, the `vmm_notify` handle and its notification identifier.

```
scoreboard sb = new();
vmm_notify_observer#(scoreboard, ahb_trans)
   observe_start = new(sb, mon.notify, mon.TRANS_START);
```

Whenever the notification event is indicated, the subscriber method (`observe_trans()`) is called.

## Transport Interfaces in OSCI TLM2.0

TLM-2.0 provides the following two transport interfaces:

- Blocking `(b_transport)`: completes the entire transaction within a single method call
- Non-blocking `(nb_transport)`: describes the progress of a transaction using multiple `nb_transport` method calls going back-and-forth between initiator and target

In general, any component might modify a transaction object during its lifetime (subject to the rules of the protocol). Significant timing points during the lifetime of a transaction (for example: start-of-response-phase) are indicated by calling `nb_transport` in either forward or backward direction, the specific timing point being given by the phase argument.

Protocol-specific rules for reading or writing the attributes of a transaction can be expressed relative to the phase. The phase can be used for flow control, and for that reason might have a different value at each hop taken by a transaction; the phase is not an attribute of the transaction object.

A call to `nb_transport` always represents a phase transition. However, the return from `nb_transport` might or might not do so, the choice being indicated by the value returned from the function (TLM_ACCEPTED versus TLM_UPDATED).

Generally, you indicate the completion of a transaction over a particular hop using the value of the phase argument. As a shortcut, a target might indicate the completion of the transaction by returning a special value of TLM_COMPLETED. However, this is an option, not a necessity.

The transaction object itself does not contain any timing information by design. Or even events and status information concerning the API. You can pass the delays as arguments to `b_transport / nb_transport` and push the actual realization of any delay in the simulator kernel downstream and defer (for simulation speed).

In summary:

- Call to `b_transport` = start-of-life of transaction
- Return from `b_transport` = end-of-life of transaction
- Phase argument to `nb_transport` = timing point within lifetime of transaction
- Return value of `nb_transport` = whether return path is being used (also shortcut to final phase)
- Response status within transaction object = protocol-specific status, success/failure of transaction

On top of this, TLM-2.0 defines a generic payload and base protocol to enhance interoperability for models with a memory-mapped bus interface.

It is possible to use the interfaces described above with user-defined transaction types and protocols for the sake of interoperability. However, TLM-2.0 strongly recommends either using the base protocol off-the-shelf or creating models of specific protocols on top of the base protocol.

## Blocking Transport

As given in the OSCI-TLM2.0 user manual,

> “The new TLM-2 blocking transport interface is intended to support the loosely-timed coding style. The blocking transport interface is appropriate where an initiator wishes to complete a transaction with a target during the course of a single function call, the only timing points of interest being those that mark the start and the end of the transaction. The blocking transport interface only uses the forward path from initiator to target.”

Due to its loosely timed application with single socket, the blocking transport interface is simpler than non-blocking transports. It only implements the forward path port called `vmm_tlm_b_transport_port` for issuing transactions and `vmm_tlm_b_transport_export` for receiving transactions

Example 5-15 shows how to build up the parent-child association during construction. It instantiates the port in the *initiator* and call the `b_transport()` method from within the port.

*Example 5-15 TLM Blocking Port Instantiation and Usage in Initiator*

```
class initiator extends vmm_xactor;
   vmm_tlm_b_transport_port#(initiator, my_trans)
      b_port = new(this,"initiator_port");
   ...
   virtual task run_ph();
      int delay;
      vmm_tlm::phase_e ph;
      ...
      // send transaction using b_transport task
      b_port.b_transport(trans, delay);
   endtask: run_ph
endclass: initiator
```

Example 5-16 shows how to instantiate the export in the *target* and implement the `b_transport()` functionality locally.

*Example 5-16 TLM Blocking Export Instantiation and Usage in Target*

```
class target extends vmm_xactor;
   vmm_tlm_b_transport_export#(target,my_trans)
      b_export = new(this,"target_export");
   ...
   task b_transport(int id = -1, my_trans trans,
                    ref int delay);
      trans.display("From Target"); //execute transaction
   endtask: b_transport
endclass: target
```

Example 5-17 shows how to instantiate the initiator/target and bind the `export` with the port.

*Example 5-17 Binding TLM Blocking Interface in Initiator and Target*

```
class my_env extends vmm_group;
   initiator initiator0; target target0;
   virtual function void connect_ph();
      //bind port -> export
      initiator0.b_port.tlm_bind(target0.b_export);
      ...
```

## Non-Blocking Transport

As given in the OSCI-TLM2.0 user manual,

> "The non-blocking transport interface is intended to support the approximately-timed coding style. The non-blocking transport interface is appropriate where it is desired to model the detailed sequence of interactions between initiator and target during the course of each transaction. In other words, to break down a transaction into multiple phases, where each phase transition marks an explicit timing point."

Both forward and backward directions are available in the non-blocking transports called `vmm_tlm_nb_transport_fw_port` and `vmm_tlm_nb_transport_fw_export` respectively. These classes are virtual and are used as a foundation for other TLM transport interfaces as described in this chapter.

Example 5-18 shows how to use a non-blocking forward port for non-blocking transportation, instantiate the port in the initiator and call the `nb_transport_fw()` API from within the port.

*Example 5-18 TLM Non-Blocking Port Instantiation and Usage in Initiator*

```
class initiator extends vmm_xactor;
   vmm_tlm_nb_transport_port#(initiator, my_trans)
      nb_port = new(this,"initiator_port");
   ...
   virtual task run_ph();
      int delay;
      vmm_tlm::phase_e ph;
       ...
      nb_port.nb_transport_fw(trans, ph, delay);
   endtask: run_ph
endclass: initiator
```

Example 5-19 shows how to instantiate the export in the target and implement the `nb_transport_fw()` functionality locally:

*Example 5-19 TLM Non-Blocking Export Instantiation and Usage in Target*

```
class target extends vmm_xactor;
   vmm_tlm_nb_transport_export#(target,my_trans,
                                vmm_tlm::phase_e)
      nb_export = new(this,"target_export");
   ...
   function vmm_tlm::sync_e nb_transport_fw(
                            int id=-1,
                            my_trans trans,
                            ref vmm_tlm ph,
                            ref int delay);
```

```
      trans.display("From Target"); //execute transaction
      return vmm_tlm::TLM_ACCEPTED; //finish completion
                                    //model
   endfunction: nb_transport
endclass: target
```

**Example 5-20 shows how to instantiate the initiator and target and bind the `nb_export` with the `nb_port`.**

*Example 5-20 Binding TLM Non-Blocking Interface in Initiator and Target*

```
class my_env extends vmm_group;
   initiator initiator0;
   target target0;
   ...
   virtual function void connect_ph();
      ...
      initiator.nb_port.tlm_bind(target0.nb_export);
         //connectivity
   endfunction: connect_ph
endclass: my_env
```

## Sockets

OSCI-TLM 2.0 uses sockets to communicate between transaction level elements. A similar set of methods is in VMM, which helps lowering the learning curve for SystemC engineers. This section describes how you can connect VMM objects to fulfill necessary communication completion models.

Sockets group together all the necessary core interfaces for transportation and binding, allowing more generic usage models than just TLM core interfaces.

OSCI-TLM 2.0 does not recommend the usage of TLM core-interfaces without sockets. However, the socket infrastructure restricts the binding model and in SystemVerilog. You need to

implement all functions even if you do not use them. You can consider this to be unnecessary as the flexibility of the core-interfaces is more suitable for verification connection models.

The `vmm_tlm_initiator_socket` and `vmm_tlm_target_socket` are generic convenience sockets ready for you to use. You can use these sockets as blocking or non-blocking transportation mechanisms.

Example 5-21 shows how to instantiate an initiator socket in the initiator and call the `nb_transport_fw` method. You must implement the backward path function `nb_transport_bw` even if it is not used, because other sockets might call this function.

*Example 5-21 Using TLM Socket for Initiator*

```
class initiator extends vmm_xactor;
   vmm_tlm_initiator_socket#(initiator, my_trans,
                                        vmm_tlm::phase_e)
   socket = new(this, "initiator_put");
   ...
   virtual function vmm_tlm::sync_e nb_transport_bw(
      int id=-1, my_trans trans,
      vmm_tlm::phase_e ph,ref  int delay);
      // Implement incoming backward path function
      return vmm_tlm::TLM_COMPLETED;//finish transaction
   endfunction: nb_transport_bw

   virtual task run_ph();
     ...
      socket.nb_transport_fw(trans, ph, delay); // Forward
path
   endtask: run_ph
endclass: initiator
```

Example 5-22 shows how to instantiate a target socket in the target and implement the `nb_transport_fw()` functionality locally. You must implement the `b_transport()` task even if it is not used because other sockets might call this task.

*Example 5-22 Using TLM Socket for Target*

```
class target extends vmm_xactor;
   vmm_tlm_target_socket#(target, my_trans,
                                  vmm_tlm::phase_e)
   socket = new(this, "target_put");

   virtual function vmm_tlm::sync_e nb_transport_fw(
      int id = -1, my_trans trans,
      ref vmm_tlm::phase_e ph, ref int delay);
      // Implement incoming forward path function
      trans.display("From Target"); //execute transaction
      return vmm_tlm::TLM_UPDATED; //finish completion
                                   //model
   endfunction: nb_transport_fw

   virtual task b_transport(int id = -1, my_trans trans,
                            ref int delay );
      ...
   endtask : b_transport
endclass: target
```

Example 5-23 shows how to instantiate the initiator and target and bind the sockets together.

*Example 5-23 Bind TLM Socket to Initiator and Target*

```
class env extends vmm_group;
   initiator initiator0;
   target target0;
   virtual function void connect_ph();
      initiator0.socket.tlm_bind(target0.socket);
      ...
   endfunction
   ...
endclass
```

## Connecting Blocking Components to Non-blocking Components

VMM provides a transport interconnect class to connect a blocking initiator to a non-blocking target or to connect a non-blocking initiator to a blocking target using the `vmm_tlm_transport_interconnect` class. This interconnect is based upon the OSCI-TLM2.0 simple socket but unlike the OSCI-TLM2.0 simple socket it does not allow blocking to blocking to blocking transport connection or non-blocking to non-blocking transport connection.

The `vmm_tlm_transport_interconnect` class uses `vmm_tlm::phase_e` as the phase type for the blocking and non-blocking TLM ports. If other user-defined phase type is required then the transport interconnect base class `vmm_tlm_transport_interconnect_base` can be used to extend the user-defined transport interconnect. You are required to implement the `b_transport, nb_transport_fw` and `nb_transport_bw` methods using the custom phases. If your phase type is different from `vmm_tlm::phase_e,` but the phase information not used in transport communication, then instantiating the `vmm_tlm_transport_interconnect_base` parameterized on the phase type, using the default implementation of `b_transport, nb_transport_fw` and `nb_transport_bw` is sufficient.

The connection between the transport port and export is done using the `tlm_bind method` of the interconnect class. For the `vmm_tlm_transport_interconnect` a `vmm_connect` utility method `vmm_transport_interconnect` is provided.

Example 5-24 shows a initiator with a TLM blocking port instantiation.

*Example 5-24 TLM Blocking Port in Initiator.*

```
class initiator extends vmm_xactor;
     vmm_tlm_b_transport_port#(initiator, my_trans)
     b_port = new(this,"initiator_port");
     ...
     virtual task run_ph();
     int delay;
     vmm_tlm::phase_e ph;
     ...
     // send transaction using b_transport task
     b_port.b_transport(trans, delay);
     endtask: run_ph
endclass: initiator
```

Example 5-25 shows a consumer with a TLM non-blocking export instantiation.

*Example 5-25 TLM Non-blocking Export in Consumer*

```
class target extends vmm_xactor;
     vmm_tlm_nb_transport_export#(target,my_trans,
     vmm_tlm::phase_e)
     nb_export = new(this,"target_export");
     ...
     function vmm_tlm::sync_e nb_transport_fw(
     int id=-1,
     my_trans trans,
     ref vmm_tlm ph,
     ref int delay);
     trans.display("From Target"); //execute transaction
     return vmm_tlm::TLM_ACCEPTED; //finish completion
                                        //model
     endfunction: nb_transport
endclass: target
```

Example 5-26 shows how to connect the initiator's blocking transport port to the target's non-blocking transport export using the `vmm_connect#(.D(d))::tlm_transport_interconnect` utility class method.

*Example 5-26 Connecting Blocking Port to Non-blocking Export*

```
class subenv extends vmm_group;
     initiator i0;
     target t0;
     ...
     virtual function void connect_ph();
     vmm_connect
     #(.D(my_trans))::tlm_transport_interconnect(
     t0.b_port,
     i0.nb_export,
     vmm_tlm::TLM_NONBLOCKING_EXPORT);
     endfunction: connect_ph
     ...
endclass: subenv
```

## Generic Payload

Generic payload is a class that has been introduced in OSCI TLM 2.0. It is primarily aimed at bus-oriented protocols, such as, AHB, OCP, etc. Generic payload contains data members such as, address, payload, command, etc. It can support other protocols with this base class by using the extension member.

You should derive a transaction from `vmm_data` to have complete control over the data object and an abstract implementation that you can reuse throughout the environment. You can use this `vmm_data` with all objects including generators and channels.

You derive the `vmm_tlm_generic_payload` from `vmm_rw_access` and use it to mainly simplify the task of bringing existing TLM SystemC generic payload objects into a VMM environment.

Example 5-27 shows the use of a generic payload, where the initiator class has a bi-directional non-blocking port parameterized on `vmm_tlm_generic_payload`.

The following initiator class has a bi-directional non-blocking port parameterized on `vmm_tlm_generic_payload`.

*Example 5-27 Using Generic Payload in Initiator*

```
class initiator extends vmm_xactor;
   vmm_tlm_nb_transport_port#(initiator,
                              vmm_tlm_generic_payload,
                              vmm_tlm::phase_e)
         nb_port = new(this,"initiator_port");
   ...
   virtual task run_ph();
      vmm_tlm_generic_payload trans;
      int delay;
      vmm_tlm::phase_e ph;
      vmm_tlm::sync_e status;
      ...
      ph = vmm_tlm::BEGIN_REQ;
      status = nb_port.nb_transport_fw(trans, ph, delay);
   endtask: run_ph

   function vmm_tlm::sync_e nb_transport_bw(int id=-1,
                            vmm_tlm_generic_payload trans,
                            ref vmm_tlm::phase_e ph,
                            ref int delay);
      ...
      ph = vmm_tlm::END_RESP;
      return vmm_tlm::TLM_COMPLETED;
   endfunction: nb_transport_bw
endclass: initiator
```

Example 5-20 shows how to model a target class that connects to the port of the initiator and that uses the `vmm_tlm_generic_payload`.

*Example 5-28 Using Generic Payload in Target*

```
class target extends vmm_xactor;
   vmm_tlm_nb_transport_export#(target,
                                vmm_tlm_generic_payload,
                                vmm_tlm::phase_e)
      nb_export = new(this,"target_export");
   …
   function vmm_tlm::sync_e nb_transport_fw(int id= -1,
         vmm_tlm_generic_payload trans,
         ref vmm_tlm::phase_e ph, ref int delay);
      trans.display("From Target"); //execute transaction
      ph = vmm_tlm::END_REQ;
      return vmm_tlm::TLM_UPDATED; //finish completion
                                   //model
   endfunction: nb_transport

   virtual task run_ph();
      ...
      nb_export.nb_transport_bw(trans, ph, delay);
   endtask: run_ph
endclass: target
```

## Broadcasting Using TLM2.0

Analysis ports are useful to broadcast transactions to multiple observers like scoreboards and functional coverage models. You can bind analysis ports to multiple observers and analysis exports to multiple producers.

As given in the OSCI-TLM2.0 manual,

"Analysis ports are intended to support the distribution of transactions to multiple components for analysis, meaning tasks such as checking for functional correctness or collecting functional coverage statistics. The key feature of analysis ports is that a single port can be bound to multiple channels or subscribers such that the port itself replicates each call to the interface method **write** with each subscriber. An analysis port can be bound to zero or more subscribers or other analysis ports, and can be unbound. Each subscriber implements the **write** method of the **tlm_analysis_if**."

## Analysis Port Usage with Many Observers

Example 5-29 shows the usage of an analysis port connected to many observers. It instantiate the `analysis_port` within the transmitter and call `write()` function.

*Example 5-29 Declaration of Analysis Port and Usage in Broadcaster*

```
class monitor extends vmm_xactor;
   vmm_tlm_analysis_port#(monitor,my_trans)
      analysis_port=new(this,"target_analysis_port");
   ...
   virtual task run_ph();
      analysis_port.write(trans); // Transmit trans to
                                  //observers
   endtask: run_ph
endclass: monitor
```

Example 5-30 shows how to instantiate the `analysis_export` within the observer and implement the `write()` functionality.

*Example 5-30 Declaration of Analysis Port and Usage in Listener*

```
class observer extends vmm_object;
   vmm_tlm_analysis_export#(observer,my_trans)
      scb_aport= new(this,"observing_analysis");
```

```
   ...
   virtual function void write(int id= -1, my_trans trans);
      trans.display("");//operation on transaction received
   endfunction: write
endclass: observer
```

Example 5-31 shows how to optionally instantiate the `analysis_export` within different observers and implement the `write()` functionality.

*Example 5-31 Multiple Analysis Port Listeners*

```
class cov_model extends vmm_object;
   ...
   vmm_tlm_analysis_export#(cov_model,my_trans)
      cov_aport= new(this,"coverage_analysis");

  covergroup covg with function sample(my_trans incoming);
      coverpoint incoming.rw;
   endgroup

   covg cg=new();
   ...
   virtual function void write(int id= -1, my_trans trans);
      this.cg.sample(trans);
   endfunction : write
endclass: cov_model
```

Example 5-32 shows how to instantiate the objects and connect the ports.

*Example 5-32 Binding Analysis Port*

```
class my_env extends vmm_group;
   ...
   monitor  mon;
   observer observe;
   cov_model cov;
   virtual function void connect_ph();
      ...
```

```
      mon.analysis_port.tlm_bind(observe.scb_aport);
      mon.analysis_port.tlm_bind(cov.cov_aport;
   endfunction : build
endclass: my_env
```

## Analysis Port Multiple Ports Per Observer

There is no restriction in OSCI-TLM2.0 to limit the number of observer hooks using `analysis_export` per observation class. However, in SystemVerilog there can only have one implementation of a function present in a class. Therefore, if you have two `analysis_exports`, these use the same `write()` implementation.

The observer might require a unique implementation of a write method for each port. Then you can instantiate multiple analysis exports in the observer with a unique implementation of write, for each binding using the shorthand macro. Alternatively, you can connect multiple ports to the same export instance using peer IDs.

## Shorthand Macro IDs

Example 5-33 shows how to use multiple `analysis_exports` within a single observer. It instantiates the `analysis_port` within the transmitter and call the `write()` function.

*Example 5-33 Declaration of Analysis Port and Usage in Broadcaster*

```
class monitor extends vmm_xactor;
   ...
   vmm_tlm_analysis_port#( monitor,my_trans)
       analysis_port = new(this,"monitors_analysis_port");
   task perform_update()
      analysis_port.write(trans);
   endtask
```

```
   ...
endclass: monitor
```

Example 5-34 shows how to instantiate two `analysis_exports` within the observer and implement the `write<ID>()` functionality.

*Example 5-34 Declaration of Multiple Analysis Ports*

```
class scoreboard extends vmm_object;
   `vmm_tlm_analysis_export(_1) //uniquifier ID
   `vmm_tlm_analysis_export(_2) //uniquifier ID

   vmm_tlm_analysis_export_1#(scoreboard,my_trans)
      scb_analysis_1=new(this,"scoreboard_analysis_1");
   vmm_tlm_analysis_export_2#( scoreboard,my_trans)
      scb_analysis_2=new(this,"scoreboard_analysis_2");
   ...
   virtual function void write_1(int id= -1, my_trans trans);
      `vmm_note(log,"From scoreboard write_1");
   endfunction: write_1

   virtual function void write_2(int id= -1, my_trans trans);
      `vmm_note(log,"From scoreboard write_2");
   endfunction: write_2
endclass: scoreboard
```

Example 5-35 shows how to instantiate the objects and bind the ports to respective places.

*Example 5-35 Binding Multiple Analysis Ports*

```
class my_env extends vmm_group;
   monitor mon[2];
   scoreboard  scb;
   ...
   virtual function void connect_ph();
      mon[0].analysis_port.tlm_bind(scb.scb_analysis_1);
      mon[1].analysis_port.tlm_bind(scb.scb_analysis_2);
   endfunction: build
endclass: my_env
```

## Peer IDs

When you use peer IDs, you need only one `write()` implementation. Within it you can identify which port is performing the access and execute the appropriate functionality.

Example 5-36 shows how to use `single_export` with `peer_id`. It instantiates the `analysis_port` within the transmitter and call the `write()` function.

*Example 5-36 Declaration of Analysis Port and Usage in Broadcaster*

```
class monitor extends vmm_xactor;
   vmm_tlm_analysis_port#( monitor, my_trans)
      analysis_port=new(this,"monitor_analysis_port");
   ...
    virtual task run_ph()
      analysis_port.write(trans);
   endtask
endclass: monitor
```

Example 5-37 shows how to instantiate one `analysis_export` within the observer and implement the `write()` functionality. You must specify maximum binding in the `analysis_export` constructor.

*Example 5-37 Using Analysis Port Peer IDs for Identifying Broadcaster*

```
class scoreboard extends vmm_object;
   vmm_tlm_analysis_export#( scoreboard,my_trans)
      scb_analysis=new(this,"scoreboard_analysis", 2, 0);
   ...
   virtual function write(int id= -1, my_trans trans);
      case(id)
         0: do_compare_from_port0(trans);
         1: do_compare_from_port1(trans);
      endcase
   endfunction
```

```
endclass: scoreboard
```

Example 5-38 shows how to instantiate the objects and bind the ports to respective places using peer IDs.

*Example 5-38 Binding Multiple Peers*

```
class my_env extends vmm_group;
   monitor  mon[2];  scoreboard  scb;
   ...
   virtual function void connect_ph();
      ...
      mon[0].analysis_port.tlm_bind(scb.scb_analysis, 0);
      mon[1].analysis_port.tlm_bind(scb.scb_analysis, 1);
   endfunction : build
endclass: my_env
```

## Interoperability Between vmm_channel and TLM2.0

VMM provides a methodology for connecting `vmm_xactors` with `vmm_xactors` using a channel interface to `vmm_xactors`.

Conversely, it is possible to connect TLM2.0 interfaces directly to `vmm_channel`. You can connect `vmm_channel` to the blocking transport interface, non-blocking forward interface, non-blocking bidirectional interface or the analysis interface.

`vmm_channel` can act as a producer by binding the channel's TLM port to an external TLM export or a consumer by binding the channel's TLM export to an external TLM port.

## Connecting vmm_channel and TLM interface

Example 5-39 shows how to connect a consumer with a `vmm_channel` to a producer with a TLM blocking port. It connects the producer with a blocking transport port calling the `b_transport` method of the blocking port.

*Example 5-39 Initiator With TLM Blocking Interface*

```
class initiator extends vmm_xactor;
   vmm_tlm_b_transport_port#(initiator,my_trans)
      b_port=new(this,"initiator_port");

    virtual task run_ph();
      ...
      b_port.b_transport(tr,delay);
   endtask: run_ph
endclass: initiator
```

Example 5-40 shows how to model target that includes a `vmm_channel` instantiated using the `vmm_channel_typed` class.

*Example 5-40 Target With Channel*

```
class target extends vmm_xactor;
   vmm_channel_typed#(my_trans) in_chan =
      new("target","in_chan");

   virtual task run_ph();
      in_chan.get(tr);
      ...
   endtask: run_ph
endclass: target
```

Example 5-41 shows how to bind the channel's blocking transport export to the blocking transport port of the initiator using the `vmm_connect#(.D(d))::tlm_bind` utility class method.

*Example 5-41 Binding Channel and TLM Blocking Interface*

```
class subenv extends vmm_group;
   initiator i0;
   target t0;
   ...
   virtual function void connect_ph();
     vmm_connect #(.D(my_trans))::tlm_bind(
         t0.in_chan,
         i0.b_port,
         vmm_tlm::TLM_BLOCKING_EXPORT);
   endfunction: connect_ph
   ...
endclass: subenv
```

## TLM2.0 Accessing Generators

VMM atomic and scenario generators have a built-in `vmm_channel` called `out_chan`. You can connect the output channel's blocking or non-blocking forward transport port to a consumer's blocking or non-blocking forward export.

Example 5-42 shows how to use an atomic generator with a consumer class that is based on a TLM blocking transport export.

*Example 5-42 Modeling a Driver With TLM Blocking Interface*

```
class driver extends vmm_xactor;
   vmm_tlm_b_transport_export#(driver,my_trans)
      b_export=new(this,"driver_export");

   task b_transport(int id=-1, my_trans trans,
                    ref int delay);
      ...
      //process the transactions received from the generator
   endtask: b_transport
endclass: driver
```

Example 5-43 shows how to instantiate the driver and atomic generator and then bind the generators blocking transport port to the driver's blocking transport export using the `vmm_connect#(.D(d))::tlm_bind` utility class method.

*Example 5-43 Binding Atomic Generator and TLM Blocking Interface*

```
class my_env extends vmm_group;
   vmm_atomic_gen #(my_trans) gen;
   driver d0;
   virtual function void connect_ph();
     vmm_connect #(.D(my_trans))::tlm_bind(
         gen.out_chan,
         d0.b_export,
         vmm_tlm::TLM_BLOCKING_PORT);
   endfunction: connect_ph

endclass: env
```

## Forward Path Non-Blocking Connection

Example 5-44 shows how to use the `vmm_channel` with a non-blocking transport connection on the forward path. The transactor with the `vmm_channel` is the producer that is connected to the non-blocking forward export of the consumer. It creates a producer class with a `vmm_channel`. The shorthand macro `` `vmm_channel `` or `vmm_channel_typed` class can be used.

*Example 5-44 Transactor With Channel*

```
class initiator extends vmm_xactor;
   vmm_channel_typed#(my_trans)
      out_chan=new("target","out_chan");

   virtual task run_ph();
      out_chan.put(tr);
      ...
   endtask: run_ph
```

```
endclass: initiator
```

Example 5-45 shows how to create a consumer class with a non-blocking forward transport export.

*Example 5-45 Target With TLM Non-Blocking Interface*

```
class target extends vmm_xactor;
   vmm_tlm_nb_transport_fw_export#(target,my_trans)
      nb_export=new(this,"target_export");

   function vmm_tlm::sync_e nb_transport_fw(int id=-1,
                                    my_trans trans,
                                    ref vmm_tlm::phase_e ph,
                                    ref int delay);
      ...
      //process the transactions received from the initiator
   endfunction: nb_transport_fw
endclass: target
```

Example 5-46 shows how to connect the non-blocking forward transport port of the channel to the non-blocking forward export of the target.

*Example 5-46 Binding Channel and TLM Non-Blocking Interface*

```
class my_env extends vmm_group;
   initiator i0;
   target t0;
   virtual function void connect_ph();
     vmm_connect #(.D(my_trans))::tlm_bind(
         i0.out_chan,
         t0.nb_export,
         vmm_tlm::TLM_NONBLOCKING_FW_PORT);
   endfunction: connect_ph
endclass: my_env
```

## Bidirectional Non-Blocking Connection

Example 5-47 shows how to connect a consumer with a `vmm_channel` to a producer with a TLM non-blocking bi-directional port. Here, a producer with a non-blocking transport port calls the `nb_transport` method of the non-blocking port.

*Example 5-47 Initiator With TLM Non-Blocking Interface*

```
class initiator extends vmm_xactor;
   vmm_tlm_nb_transport_port#(initiator,my_trans)
      nb_port=new(this,"initiator_port");

   virtual task run_ph();
      ...
      nb_port.nb_transport_fw(tr,ph,delay);
   endtask

   function vmm_tlm::sync_e nb_transport_bw(int id=-1,
                                            my_trans trans,
                                    ref vmm_tlm::phase_e ph,
                                            ref int delay);
      //method is called when target notifies
      //vmm_data::ENDED on a particular transaction
   endfunction
endclass: initiator
```

Example 5-48 shows how to model a target with a `vmm_channel` instantiated using the `vmm_channel_typed` class.

*Example 5-48 Target With Channel*

```
class target extends vmm_xactor;
   vmm_channel_typed#(my_trans)
      in_chan=new("target","in_chan");
   virtual task run_ph();

      in_chan.get(tr);
      ...
```

```
      tr.notify.indicate(vmm_data::ENDED); //calls the
         //nb_transport_bw method of the initiator with
         //the current transaction
   endtask: run_ph
endclass: target
```

Example 5-49 shows how to bind the channel's non-blocking bi-directional export to the non-blocking bi-directional port of the initiator using the `vmm_connect#(.D(d))::tlm_bind` utility class method.

*Example 5-49 Binding Channel and TLM Non-Blocking Interface*

```
class subenv extends vmm_subenv;
   initiator i0;
   target t0;
   virtual function void connect_ph();
     vmm_connect #(.D(my_trans))::tlm_bind(
         t0.in_chan,
         i0.nb_port,
         vmm_tlm::TLM_NONBLOCKING_EXPORT);
   endfunction: connect_ph
endclass: subenv
```

# Advanced Usage

## Updating Data in Analysis Ports From vmm_notify

VMM has a default subscription based listener model based on `vmm_notify`. You can use VMM notification service (`vmm_notify`) to connect a transactor, a channel, or any other testbench component to a scoreboard or functional coverage collector or any other passive observer. There can be multiple observers, and they will all see the same transaction stream.

There are pre-defined notification in `vmm_xactor` and `vmm_channel` readily available for review and use.

Example 5-50 shows how to configure your notification normally and call the `indicate()` API as usual.

*Example 5-50 Modeling Monitor With Notification*

```
class monitor extends vmm_xactor;
   ...
   int OBSERVED;
   function new(string name);
      this.OBSERVED=this.notify.configure();
   endfunction

   virtual task run_ph()
      ...
      this.notify.indicate(this.OBSERVED, my_trans)
   endtask: run_ph
   ...
endclass
```

Example 5-51 shows how to implement the `indicated()` functionality to pass the transaction onto observer.

*Example 5-51 Modeling Subscriber With Notification Callbacks*

```
class subscribe extends vmm_notify_callbacks;
   ...
   local observer obs;
   function new(observer obs);
      this.obs = obs;
   endfunction
   virtual function void indicated(vmm_data status);
      this.obs.observe(status);
   endfunction
   ...
endclass
```

Example 5-52 shows how the observer class implements the `observe()` function which executes the `analysis_port.write()`.

*Example 5-52 Modeling Observer With Ad-Hoc Analysis Port*

```
class observer extends vmm_object;
   vmm_tlm_analysis_port#(subscribe, my_trans)
      analysis_port = new(this,"observer_analysis_port");
   string name;

   function new(string name, vmm_notify ntfy, int id);
      subscribe cb = new(this);
      ntfy.append_callback(id, cb);
      this.name = name;
   endfunction

   function void observe(vmm_data tr);
      analysis_port.write(tr);
   endfunction
endclass
```

Finally, you instance the objects and bind the analysis port to any subscribing `analysis_export`. Thus, when `vmm_notifier` indicates the data object, `analysis_exports` observes it.

## Connect Utility (vmm_connect)

You can use VMM connect utility class `vmm_connect` for connecting channels and notifications in the `vmm_group::connect_ph()` method. Additionally, it checks whether you have already connected the channels to a producer and a consumer. You usually connect with the `vmm_channel set_consumer()` and `set_producer()` methods.

```
class vmm_connect #(type T, type N=T, type D=vmm_data)
```

The `vmm_connect` class has the following methods that you can use for channel/notification connectivity.

```
class vmm_connect#(T)::channel(ref T upstream, downstream,
    string name= “”, vmm_object parent = null);
```

Example 5-53 shows how to use `vmm_connect#(T)::channel()` method to connect the channels.

*Example 5-53 Connecting Producer/Consumer Channels Using vmm_connect*

```
class ahb_unit extends vmm_group;
   ahb_trans_channel gen_chan;
   ahb_trans_channel drv_chan;

   virtual function void build_ph();
      drv_chan = new(“ahb_chan”, “drv_chan”);
      gen_chan = new(“ahb_chan”, “gen_chan”);
   endfunction

   virtual function void connect_ph();
      vmm_connect#(.T(ahb_trans_channel))::channel(
         gen_chan, drv_chan, "gen2drv", this);

   endfunction

endclass
```

You should not attempt to connect two channels that are already connected together or to another channel.

Example 5-54 shows how to use the `vmm_connect#(T,N,D)::notify()` method to connect notification to the subscriber such as, scoreboard.

*Example 5-54 Using vmm_connect::notify()*

```
class scoreboard;
   virtual function void observe_trans(ahb_trans tr);
      ...
   endfunction
endclass
`vmm_notify_observer(scoreboard, observe_trans)

class ahb_unit extends vmm_group;
   scoreboard sb;

   virtual function void build_ph();
      sb = new();
   endfunction

   virtual function void connect_ph();
      vmm_connect#(.N(scoreboard), .D(ahb_trans))::notify(
          sb, mon.notify, mon.TRANS_STARTED);
   endfunction
endclass
```

## Channel Non-Atomic Transaction Execution

Non-atomic transactors execute transactions in parallel, pipelined through multiple attempts, multiple partial sub-transactions or a transaction repeatedly at regular intervals.

Such transactors use a non-blocking completion model. As shown in Figure 5-4, the execution thread from the upstream transactor (depicted as a dotted line) is not blocked while the transaction descriptor flows through the channel and the downstream transactor executes it.

It is blocked only when the channel is full and unblocks as soon as it is non-full, regardless of whether the transaction is complete or not.

The non-blocking completion model allows submission of several transactions to the downstream transactor for completion in future. It is up to the upstream transactor to detect the completion of a transaction according to a mechanism the downstream transactor defines.

The suitability and proper implementation of this completion model requires that the downstream transactor adheres to the following guidelines:

The channel instance is responsible for blocking the execution of the `vmm_channel::put()` method, not the downstream transactor. That blocking only happens if the channel is considered full.

More than one transaction must be available in the channel to allow out-of-order execution.

If you use a full level of a channel, you create a blocking interface. Non-atomic execution is only possible if the downstream transactor implements additional transaction descriptor buffering internally.

You receive the additional status information as a separate status descriptor derived from `vmm_data` and attached to the `vmm_data::ENDED notification by the vmm_channel::complete()` method.

*Example 5-55 Returning Status Information Through the Ended Notification*

```
class transaction_resp extends vmm_data;
   ...
endclass: transaction_resp

class consumer extends vmm_xactor;
   ...
   virtual task main();
      ...
      forever begin
         transaction tr;
```

```
            ...
            this.in_chan.start(tr);
            ...
            begin: status
               transaction_resp tr_status = new(...);
               ...
               this.in_chan.complete(tr_status);
            end
            ...
         end
      endtask: main
   endclass: consumer
```

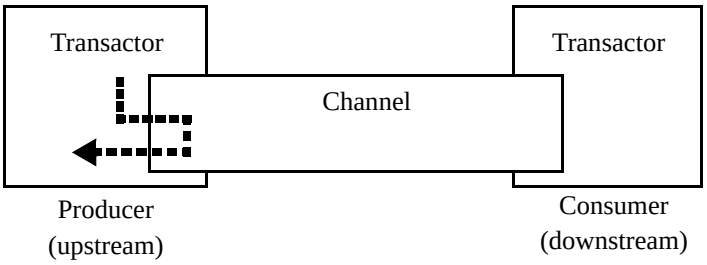
## Channel Out-of-Order Atomic Execution Model

Transactors with an out-of-order atomic execution model execute individual transactions in a potentially different order than you submit them.

The order in which you select transactions for execution is protocol-specific and out of the scope of this book. Such transactors use a non-blocking completion model.

As shown in Figure 5-4, you do not block the execution thread from the producer transactor (depicted as a dotted line) while the transaction descriptor flows through the channel and the consumer transactor executes it. You block it only when the channel is full and it unblocks as soon as the channel is empty, regardless of whether the transaction is complete or not.

*Figure 5-4 Non-Blocking Completion Model*




The non-blocking completion model allows submission of several transaction descriptors to the consumer transactor for completion in the future.

If needed, it is up to the producer transactor to detect the completion of a transaction by waiting for the indication of the *vmm_data::ENDED* notification in the transaction descriptor or the *vmm_channel::ACT_COMPLETED* indication in the input channel, as shown in Example 5-56.

*Example 5-56 Upstream of a Non-Blocking Completion Model*

```
class producer extends vmm_xactor;
   ...
   virtual task main();
      ...
      ... begin
         transaction tr;
         ...
         out_chan.put(tr);
         fork
            begin
               automatic transaction w4tr = tr;
               w4tr.wait_for(vmm_data::ENDED);
               ...
            end
         join_none
         ...
      end
   endtask: main
```

```
   ...
endclass: producer
```

The suitability and proper implementation of this completion model requires that consumer transactors adhere to the following guidelines:

Out-of-order transactors often execute transactions in a sequence other than the one you submit because they implement different priorities or class of services for different transactions.

If a transactor offers more than one execution priority or class of service, it must use a different input channel for each. Using a single channel might block the execution of higher priority transactions because you fill it with low-priority transactions.

You assume the transactions in the channel to be available for execution. As soon as you select a transaction for execution (concurrently, partially or as the first instance of a recurrence), you might immediately remove it from the channel to prevent it from being selected again by another transaction execution thread. You need this if the channel is connected to multiple consumers.

*Example 5-57 Removing a Transaction Descriptor From the Input Channel*

```
class consumer extends vmm_xactor;
   ...
   virtual task main();
      ...
      forever begin
         ...
         this.in_chan.get(tr);
         tr.notify.indicate(vmm_data::STARTED);
         ...
      end
   endtask: main
endclass: consumer
```

A producer transactor might track individual transactions by maintaining a reference to the transaction descriptors as they flow through the channel and the downstream transactor executes them. You use the `vmm_notify::indicate` function already in the transaction descriptor to eliminate the need for additional synchronization infrastructure in the upstream transactor.

*Example 5-58 Indicating Transaction Execution Notifications*

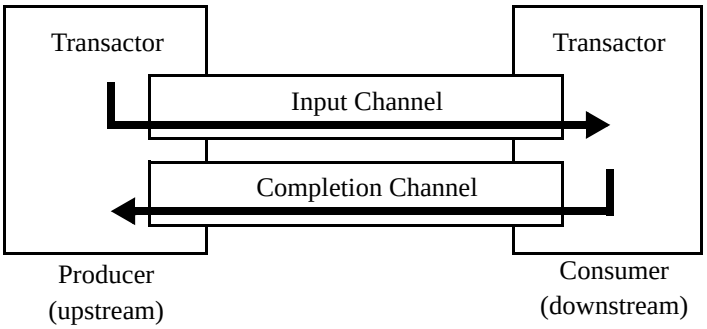
```
class consumer extends vmm_xactor;
   ...
   virtual task main();
      ...
      while (1) begin
         transaction tr;
         this.in_chan.get(tr, i);
         tr.notify.indicate(vmm_data::STARTED);
         ...
         tr.notify.indicate(vmm_data::ENDED);
      end
   endtask: main
   ...
endclass: consumer
```

You cannot use the `vmm_channel::active()`, `vmm_channel::start()`, `vmm_channel::complete()` and `vmm_channel::remove()` methods because they support an atomic i.e. one at a time execution model. You cannot use these methods when you execute multiple transactions concurrently.

A producer transactor might require information about the various intermediate completions of a transaction execution such as each execution attempt, each sub-transaction or each occurrence of a recurring transaction.

As a transaction might have more than one completion indication, you should use an output channel to return completion information back to the producer transactor, as shown in Figure 5-5.

*Figure 5-5 Completion Channel*




This usage avoids stalling the consumer transactor on a full completion channel when the producer transactor fails to drain it. No data is lost even if the channel becomes full.

*Example 5-59 Providing Completion Status Through Completion Channel*

```
class consumer extends vmm_xactor;
   transaction_channel      in_chan;
   transaction_resp_channel compl_chan;

   virtual task main();
      ...
      forever begin
         ...
         this.in_chan.get(tr);
         tr.notify.indicate(vmm_data::STARTED);
         ...
         begin
            transaction_resp resp = new(...);
            tr.notify.indicate(vmm_data::ENDED, resp);
            this.compl_chan.sneak(resp);
         end
      end
   endtask: main
endclass: consumer
```

When you can use the transaction descriptor 's properties to specify completion status information, you modify these properties by the consumer transactor to provide status information back to the producer transactor.

A single transaction descriptor might result in multiple completion responses back through the completion channel. When you use the same instance, subsequent responses might modify the content of prior responses before the producer transactor has time to process them.

Using separate instances for each response ensures that you receive an accurate report of the history of the transaction execution via the completion channel.

If the transaction descriptor does not have properties that you can use to specify completion status information, the consumer transactor can provide status information back to the upstream transactor via a different status descriptor supplied through the completion channel.

You provide additional status information as a separate descriptor derived from `vmm_data`. You should provide a reference to the original transaction in the status descriptor. It is not necessary to overload all of the virtual methods in the status information class. This is shown in Example 5-59.

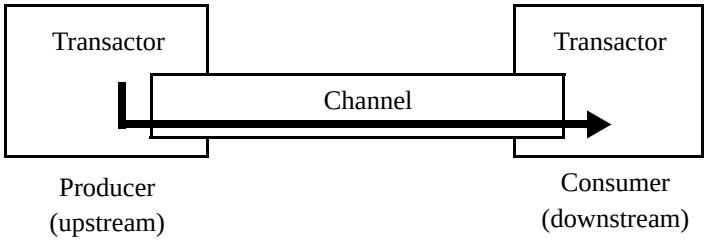
## Channel Passive Response

Passive transactors monitor transactions executed on a lower-level interface and report to the higher-layers descriptions of the observed transactions.

A passive transactor should report any protocol-level errors it detects. However, the higher-level transactors are responsible for checking the correctness of the data carried by the protocol. As

shown Figure 5-6, passive transactors use an output channel to report transactions. Using a new instance of the transaction descriptor, you report each observed transaction.

*Figure 5-6 Passive Response Model*




Note: You do not limit the passive response model to passive transactors. You can use it to report on observed transactions in various transactors. A reactive transactor might use the passive response model to report on the observed transactions that received active replies. A proactive transactor might use a passive response model to report on the received transactions as observed on a half-duplex interface.

The suitability and proper implementation of this response model requires that passive transactors adhere to the following guidelines:

The output channel will block the execution thread of the passive transactor if it becomes full. This blocking might break its implementation or cause data to be lost.

The `vmm_channel::sneak()` method ignores the channel's full level and never blocks the execution thread of the upstream transactor. Because the passive monitor is observing the proper execution of a protocol, you should regulate its execution by the time required to execute a complete transaction.

A consumer transactor might need to know when a transaction has started execution on an interface. For example, a half-duplex higher-level transactor would need to know if the transport medium is busy before attempting to execute its own transaction. Waiting until the end of the transaction to put it in the output channel might delay the information much.

*Example 5-60 Incomplete Transaction Descriptor in an Output Channel*

```
class producer extends vmm_xactor;
   ...
   virtual task main();
      ...
      while (1) begin
         ...
         tr = new;
         ...
         tr.notify.indicate(vmm_data::STARTED);
         this.out_chan.sneak(tr);
         ...
         tr.notify.indicate(vmm_data::ENDED);
      end
   endtask: main
endclass: producer
```

Consumer transactors can also use the timestamps associated with these notifications for identifying time-related information about the transaction such as its total execution time.

*Example 5-61 Monitoring Transactions From a Passive Transactor*

```
class consumer extends vmm_xactor;
   ...
   virtual task main();
      ...
      while (1) begin
         ...
         this.in_chan.peek(tr);
         tr.notify.wait_for(vmm_data::ENDED);
         this.in_chan.get(tr);
         ...
      end
   endtask: main
```

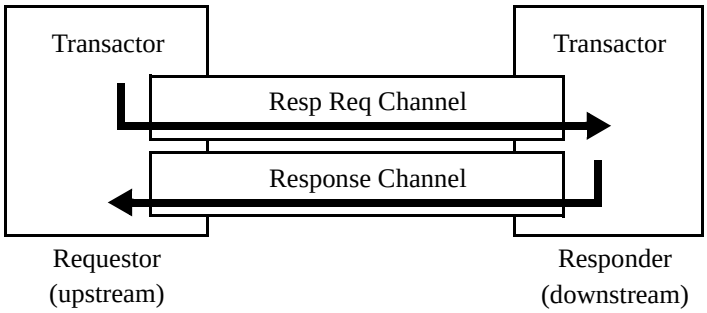
```
endclass: consumer
```

## Channel Reactive Response

Reactive transactors monitor the transactions executed on a lower-level interface and might have to request additional data or information from higher-layer transactors to complete the transaction. Reactive transactors should report any protocol-level errors detected and locally generate protocol-level answers. However, higher-level transactors are responsible for providing correct data content to be carried by the protocol.

As shown in Figure 5-7, reactive transactors use an output channel to request a transaction response. A second input channel is used to receive the transaction response applied to the lower-level interface. Each transaction response request is reported using a new instance of a transaction response descriptor object.

*Figure 5-7 Reactive Response Model*




Note: You only use the reactive response model to obtain higher-level data the protocol carries. Where the protocol fully defines the entire set of possible responses, the reactive transactor internally generates the response.

For example, deciding to reply to a USB transaction with an ACK, NACK, STALL packet or not replying at all can be entirely decided internally. However, a reactive response model should provide the content and length of a *DATA* packet in reply to an IN transaction. Note you provide the response within sufficient time to avoid breaking the protocol.

The suitability and proper implementation of this response model requires that reactive transactors adhere to the following guidelines.

The implementation of the protocol might require that the requestor transactor performs additional operations while the response is being “composed.” The `vmm_channel::sneak()` method ensures that the requestor transactor execution is never blocked, if only to immediately wait for a response via the response channel.

*Example 5-62 Requesting a Response*

```
class requestor extends vmm_xactor;
   ...
   virtual task main();
```

```
      ...
      forever begin
         ...
         resp = new;
         ...
         this.req_chan.sneak(resp);
         ...
         this.resp_chan.get(resp);
         ...
      end
   endtask: main
endclass: requestor
```

You usually limit the time required to respond to a transaction by the lower-level protocol specification. However, the requestor transactor controls the time required to “compose” the response. Thus, the requestor transactor can only check that the response comes back when required.

*Example 5-63 Checking Response Request Fulfillment Delay*

```
class requestor extends vmm_xactor;
   ...
   virtual task main();
      ...
      forever begin
         ...
         resp = new;
         ...
         this.req_chan.sneak(resp);
         resp = null;
         fork
            this.resp_chan.get(resp);
            #(...);
         join_any
         disable fork;
         if (resp == null) ...
         ...
      end
   endtask: main
endclass: responder
```

To simplify the usage model of a reactive transactor, you might use a default response if a higher-level transactor fails to provide an explicit transaction response in time.

The higher-level transactor might have preferred to continue with the default response. However, it should issue a message to inform an unwary you of a potential problem with the verification environment.

The responding reactive monitor should fill in the content of a transaction response descriptor. By default, it should provide a random, but valid, response. Therefore, you should design the transaction response descriptor to provide a valid response when you use the `randomize()` method. You could user-extend the transaction response request descriptor to provide a more constrained response or procedurally filled in to provide a directed response.

*Example 5-64 Providing a Random Response*

```
class responder extends vmm_xactor;
   ...
   virtual task main();
      ...
      forever begin
         this.req_chan.get(tr);
         ...
         tr.stream_id = this.stream_id;
         tr.data_id   = response_id++;
         if (!tr.randomize()) ...
         ...
         this.resp_chan.sneak(tr);
      end
   endtask: main
endclass: responder
```

The protocol fully defines protocol-level responses and the reactive transactor can select without any input required from higher-level transactors.

The embedded factory-pattern generator should generate a response. By default, you constrain the generator to produce the best possible response. However, you can unconstraint or modify to respond differently or inject errors.

To ease the creation of verification environments, a reactive transactor might be configurable to generate the complete protocol response internally. This instead of deferring the higher-level data to higher-level reactive transactors.

A transactor detects whether you have provided a response within an acceptable time and determines that the response request is still in the request channel. It might then assume there are no higher-level transactors and choose to compose a default response on its own.

## vmm_tlm_reactive_if

VMM provides a methodology to facilitate writing reactive transactors using a polling approach rather than an interrupt approach. The reactive interface should be instantiated in a consumer transactor to connect to multiple producers.

It provides blocking and non-blocking (forward and bi-directional) transport exports and can be bound to more than one transport port.

The `q_size` parameter specifies how many transactions can be pending. The reactive interface provides blocking and non-blocking, `get()` and `try_get()`, methods to receive transaction on a first in first out basis. You indicate completion of the active transaction by calling the `completed()` method.

Note: You must process one transaction at a time. An error is issued if get is called before completing the previous transaction.

If the queue of pending transactions is full, all incoming transactions from non-blocking ports are refused by immediately returning the `vmm_tlm::REFUSED` status.

For blocking ports, the following behavior is observed by the initiator if the queue is full:

- For `vmm_tlm_generic_payload` transactions, the `m_response_status` field is set to `TLM_INCOMPLETE_RESPONSE` and a warning is issued. The `b_transport()` method returns immediately.
- If transactions are not of `vmm_tlm_generic_payload` type, then they continue to be queued internally passed the maximum queue size and a warning is issued. The `b_transport()` method will be blocked until the transaction is completed. If the queue of pending transaction grows to twice its maximum size, then an error is issued and the `b_transport()` method returns immediately.

If transactions can be queued, blocking initiators are blocked until the transaction is completed and non-blocking initiators are accepted by returning the `vmm_tlm::ACCEPTED` status. Pending transactions are returned to the target by the `try_get()` or `get()` methods in order of arrival.

Example 5-65 shows how to connect a TLM blocking port to reactive class.

*Example 5-65 Producer With TLM Blocking Interface*

```
class producer extends vmm_xactor;
  vmm_tlm_b_transport_port#(producer) b_port = new(this,
"producer port");
```

```
  virtual task run_ph();
    …
    b_port.b_transport(tr,delay);
  endtask: run_ph
endclas: producer
Consumer with TLM reactive interface
class consumer extends vmm_xactor;
   vmm_tlm_reactive_if#(my_trans, 4) reac_export1 =
new(this, "export1");
   virtual task run_ph();
      my_trans trans;
      fork
        while (1)
        begin
            reac_export1.get(trans);
            reac_export1.completed();
        end
      join_none
   endtask : run_ph
endclass : consumer
Binding reactive interface and TLM Blocking interface
class my_env extends vmm_group;
  producer p1;
  producer p2;
  consumer c;
  function void connect_ph();
       c.reac_export1.tlm_bind(p.b_port,
vmm_tlm::TLM_BLOCKING_EXPORT);
       c.reac_export1.tlm_bind(p.b_port,
vmm_tlm::TLM_BLOCKING_EXPORT);
 endfunction
endclass
```

# 6

# Implementing Tests & Scenarios

This chapter contains the following sections:


- Overview
- Generating Stimulus
- Modeling Scenarios
- Modeling Generators
- Implementing Testcases

## Overview

The verification planning process outlined in Chapter 2 of the VMM book produces the following three distinct sets of requirements:

- functional coverage
- stimulus generation
- response checking

This chapter focuses on the stimulus generation requirement.

This chapter is of interest to those responsible for creating reusable test scenarios and testcases through directed or random stimulus.

Directed stimulus can be considered as a subset of random stimulus and with a properly designed random generator, which can be created simply. Random generators are aimed at exercising the DUT according to the requirements outlined in the verification planning process (VMM Book, Chapter 2).

Random generators should be controllable to cover the entire spectrum of randomness between pure random and directed stimulus.

## Generating Stimulus

In a typical simulation, thousands of data items or transaction descriptors are created, which flow through transactors, record and compare in the self-checking structure. Also, only a handful of data and transaction sources that need to exist at the beginning of the simulation and remain in existence until the end are there.

You should model the generation of data (packets, frames, instructions) or transaction descriptors separately from the data models themselves because of the different dynamics of their respective lifetimes.

Generation can be a manual or directed process, where transaction descriptors and data items are individually created and submitted to the appropriate transactor.

Generation can be automated with the use of independent random generators, using *randomness* approximates automation. Left to run for long enough, a random source will on its own eventually generate the stimulus necessary to exercise a large portion of the functionality you need to verify.

Random generators succeed in their task within reasonable time. You do not ask them to replicate the exact directed stimulus an engineer has written to exercise a specific feature. Rather, you should expect random generators to hit any one of a large number of features through non-optimal random stimulus sequences.

However, pure random stimulus, which is constrained to be valid, is rarely useful. You must define the degrees of freedom in random stimulus up front to create a mix of random but interesting scenarios.

Though many verification engineers are more familiar with directed stimulus than random stimulus, random stimulus should be present first. It is difficult to evolve from a directed stimulus process to an automated, random stimulus one. However, you can consider directed stimulus a subset of, or a highly constrained random stimulus.

Random-based verification environment can be constrained or override to produce directed stimulus. Accomplishing the opposite is more difficult. If the directed stimulus only concerns a subset of the input paths to the DUT, you can use the random stimulus on the other input paths to provide background noise.

## Random Stimulus

Random stimulus is traditionally used to generate background noise. However, it should be used in lieu of directed stimulus to implement the bulk of the testbenches. Coupled with functional coverage to identify if the random stimulus has exercised the required functionality, it uses constraints to direct the generation process in appropriate corner cases.

This section specifies guidelines on how to write autonomous generators that create a stream of random data or transaction descriptors.

You should design generators to be easily externally constrained without requiring modifications of their source code. You then write constrained-random tests - not by writing a completely new or slightly modified generator - but by adding constraints and scenario definitions to the reusable generators that already exist.

Some predefined atomic and scenario generators are available in the VMM Standard Library. You can then use the `vmm_atomic_gen()` and `vmm_scenario_gen()` macros to automatically create generators that follow all guidelines outlined in this section for any user-defined type.

The Multi Stream Scenario Generator (MSSG) `vmm_ms_scenario_gen()` provides the capability to implement hierarchical and reusable transaction scenarios. It controls and coordinates existing scenarios to achieve a fine-grained control over stimulus.

As such, all guidelines applicable to transactors are applicable to generators unless explicitly superseded in this section.

*Example 6-1 Generators are Transactors*

```
class eth_frame_gen extends vmm_xactor;
   ...
endclass: eth_frame_gen
```

A generator is a transactor, which has one or more output channels. It might have input channels in the case of reactive stimulus generation.

A generator produces streams of data or transaction descriptors that need to be executed by the transactors. To connect the output of a generator to the input of a transactor, both must use the same transaction interface mechanism.

If a generator produces concurrent stimulus for multiple streams, it must have an output channel for each of the output streams. This channel connects each stream to their respective execution transactors.

For the MSGG, the channel is a logical channel, which you can dynamically bind to registered physical channels that might exist anywhere in the environment.

This structure allows several important operations that you require to implement testcases or build verification environments. For example, you can,

- query the channel, control or reconfigure it.
- reference the channel as the input channel for a downstream transactor.
- replace the channel if you require dynamic environment reconfiguration.

*Example 6-2 Generator Output Channel Class Property*

```
class eth_frame_gen extends vmm_xactor;
   ...
   eth_frame_channel out_chan;
   ...
endclass: eth_frame_gen
```

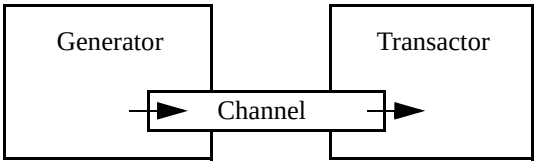
If the channel instance is not specified, then it can be instantiated as the output channel in the constructor. If it is specified, then its reference is stored in the appropriate public class property.

*Example 6-3 Connecting a Generator to a Specified Channel Instance*

```
class eth_frame_gen extends vmm_xactor;
   eth_frame_channel out_chan;
   ...
   function new(...,
                eth_frame_channel out_chan = null);
      ...
      if (out_chan == null) out_chan = new(...);
      this.out_chan = out_chan;
      ...
   endfunction: new
   ...
endclass: eth_frame_gen
```

Connecting a generator to a transactor requires that the output channel of the generator be the input channel of the downstream transactor. You can accomplish this connection if they share references to a single channel instance.

*Figure 6-1 Connecting a Generator to a Transactor*




The steps to connect a generator to a transactor are,

1. Connect one of them internally to instantiate its channel
2. Pass a reference to that channel to the constructor of the other one

Using the `vmm_connect` class for connection of channels to each other is recommended.

*Example 6-4 Instantiating the Generator First (Explicitly Phased Environment)*

```
class tb_env extends vmm_env;
   ...
   eth_frame_gen gen;
   eth_mac       mac;
   ...
   function void dut_env::build();
      this.gen = new(...);
      this.mac = new(..., this.gen.out_chan);
   endfunction: build
endclass: tb_env
```

*Example 6-5 Instantiating the Transactor First (Explicitly Phased Environment)*

```
class tb_env extends vmm_env;
   ...
   eth_frame_gen gen;
   eth_mac       mac;
   ...
   function void dut_env::build();
      this.mac = new(...);
      this.gen = new(..., this.mac.tx_chan);
   endfunction: build
endclass: tb_env
```

Alternatively, you can instantiate a stand-alone channel and then passed to the constructor of the generator and the transactor.

*Example 6-6 Instantiating the Channel*

```
class tb_env extends vmm_env;
   ...
   eth_frame_channel gen_to_mac;
   eth_frame_gen     gen;
   eth_mac           mac;
   ...
   function void dut_env::build();
      eth_frame_channel gen_to_mac =new(...);
      eth_frame_gen     gen = new(..., this.gen_to_mac);
      eth_mac           mac = new(..., this.gen_to_mac);
   endfunction: build
endclass: tb_env
```

The factory enabled transaction object is a necessity to obtain highly controllable stimulus. See “Class Factory Service” on page 25 for the process to enable and use the factory service.

You should make the prototype or blueprint for the factory a class property of the generator. It should follow a naming convention to make it easier to identify the location, name and type of all randomized instances in a verification environment. It also helps in clearly identifying the purpose of the class property.

For example, in the predefined VMM atomic generator the instance name of the prototype transaction is *`randomized_obj`*.

If a contradiction in a set of constraints makes it impossible for the solver to find a solution, the *`randomize()`* method returns non-zero.

It is important that an error is reported to indicate the problem with the constraints in the status of the simulation and to prevent using a partial solution.

*Example 6-7 Checking the Success of Randomization Process*

```
if (!this.randomized_fr.randomize()) begin
   `vmm_error(this.log, "Unable to find a solution");
   continue;
end
```

The stream identifier class property is defined in the *vmm_data* base class and inherits by all data and transaction descriptor classes.

Example 6-8 shows how to set the value of the stream identifier class property. It should be set before every randomization attempt, to ensure that the user does not accidentally modify the stream identifier in the randomized instance. It also ensures that the stream identifier is set consistently even if the randomized instance is substituted with another instance (for example, using the factory service).

*Example 6-8 Setting the stream_id Class Property*

```
while (...) begin
   ...
   this.randomized_fr.stream_id = this.stream_id;
   ...
   if (!this.randomized_fr.randomize()) ...
   ...
end
```

You might use this stream identifier to specify stream-specific constraints when adding constraints using a mechanism that is global to all instances as shown in Example 6-9.

*Example 6-9 Specifying Constraints on a Subset of Streams*

```
constraint eth_frame::tc1 {
   ...
   if (stream_id == 2) {
      ...
   }
}
```

## Directed Stimulus

Directed stimulus is manually constructed to verify a specific feature of the design or to hit a specific functional coverage point. Not all of the stimulus needs to be directed.

Random values can be used to fill portions of the stimulus that are not directly relevant to the feature being exercised. For example, the content of a packet payload is irrelevant to the correctness of the packet routing. The only requirement is that it to be transferred unmodified.

Similarly, the content and identity of the general purpose registers used in an ADD instruction is not relevant as long as the destination register eventually contains the accurate sum of the values contained in the two source registers.

You might also use random stimulus as background noise on the interfaces, not directly related to the feature you are verifying. The directed stimulus is focused on the interfaces directly implicated in the verification of the targeted functionality.

Similarly, directed stimulus might be injected in the middle of random stimulus. This sequence might help identify problems that might not be apparent, should the directed stimulus be applied from the reset state.

Directed stimulus is typically meant to replace random stimulus, not intermix with it. If the random generator is still running while directed stimulus are injected into its output stream, the resulting stimulus sequence is unpredictable.

Generators might be stopped for the duration of the simulation, while others providing background noise, might keep running as usual. Generators might be stopped at some points during the simulation, and then restart after you inject the directed stimulus.

The built-in scenario and multi stream generators provide capabilities to intermix directed and random stimulus. They reserve channels for robustness and consistency in intermixing streams of data. This is discussed in the later sections of this chapter.

*Example 6-10 Stopping a Generator at the Beginning of a Simulation*

```
class test_directed extends vmm_test;
...
vmm_xactor host_src_gen0, phy_src_gen1;
virtual function start_of_test_ph;
   ...
   $cast(this.host_src_gen0,
     vmm_object::find_object_by_name("host_src"));
   $cast(this.phy_src_gen1,
     vmm_object::find_object_by_name("phy_src"));
   this.host_src_gen0.stop_xactor();
endfunction
virtual task run_ph;
   fork
      directed_stimulus;
   join_none
endtask

task directed_stimulus;
   ...
endtask: directed_stimulus
endclass: test_directed
```

Directed stimulus can be specified by manually instantiating data and transaction descriptors and then setting their properties appropriately.

When injected in the output stream, the data or transaction descriptor is passed to the callback methods before adding them to the generator output channel. The procedure returns when the directed data has been consumed by the output channel.

*Example 6-11 Directed Transaction Interface*

```
class eth_frame_gen extends vmm_xactor;
   eth_frame_channel out_channel;
   ...
   task inject(eth_frame fr,
               ref bit dropped);
      dropped = 0;
      `vmm_callback(eth_frame_gen_callbacks,
                    post_inst_gen(this, fr, dropped));
      if (!dropped) this.out_chan.put(fr);
   endtask: inject
endclass: eth_frame_gen
```

Directed stimulus can easily be injected in the output stream of the generator by directly putting instances of transaction descriptors in the output channel. You accomplish this stimulus introduction by calling the *vmm_channel::put()* method directly.

*Example 6-12 Injecting a Directed Sequence*

```
task directed_stimulus;
   eth_frame to_phy, to_mac;
   ...
   to_phy = eth_frame::create_instance(this,"to_phy");
   to_phy.randomize();
   ...
   fork
      this.host_src_gen0.inject(to_phy, dropped);
      begin
         // Force the earliest possible collision
        @ (posedge this.vif.tx_en); //virtual interface
         this.phy_src_gen1.inject(to_mac, dropped);
      end
   join
   ...

endtask: directed_stimulus
```

It is necessary that the directed stimulus is familiar with the transactor completion model to identify when the transaction execution completes.

Further, such stimulus might not be passed to the callbacks methods of the generator and the scoreboard or the functional coverage model might not record it.

You should use this mechanism only if it is necessary to create an out-of-order or partial-execution directed stimulus. The reference to the output channel of a generator is public to allow for dynamic reconfiguration of an environment and to connect it to a downstream transactor. It is not its primary purpose to allow direct injection of directed stimulus.

## Generating Exceptions

By default, transactors execute transactions without errors, as fast as possible. However, the verification of a design necessitates that the limits of a protocol are stretched and sometimes broken. A verification environment and the transactors that compose it must provide a mechanism for injecting exceptions in the execution of a transaction.

As described in “Transactor Callbacks” on page 26, you can use the callback mechanism to cause a transactor to deviate from its default behavior.

You can inject within a callback, protocol exceptions such as, extra delays, negative replies or outright errors without modifying the original transactor. You can define many exceptions and implement in the callback methods themselves such as, inserting delays or corrupting the information in the transaction descriptor.

You must implement some exceptions in the transactor itself, such as ignoring an entire transaction or prematurely terminating a transaction. In the latter case, callback methods provide the necessary control mechanism to trigger them.

Directed exception injection is performed by extending the appropriate callback for the appropriate transactor within the testcase implementation. Then this callback is prepended to the appropriate transactor callback registry. As shown in Example 6-13, a directed testcase uses the callback mechanism to force a collision on all input ports of an ethernet device by aligning the transmission of the next frame in all MII transactors.

*Example 6-13 Aligning the Transmissions in All MII Transactors*

```
class align_tx extends mii_mac_layer_callbacks;
   local int waiting = 0;
   local int until_n = 1;
   local event go;
   ...
   virtual task pre_frame_tx(...);
      waiting++;
      if (waiting >= until_n) ->go;
      else @(go);
      waiting--;
   endtask: pre_frame_tx
enclass: align_tx
class test extends vmm_test;
virtual function connect_ph;
 begin
   align_tx cb = new(...);
   //attach callbacks using transactor iterator
   `foreach_vmm_xactor(mii_xactor, "/./", "/./") begin
      xact.prepend_callback(cb);
   end
 end
 endfunction
endclass
```

Random stimulus is proving to be a powerful mechanism to improve the productivity of functional verification. However, stimulus means more than primary data and transactions. It also includes protocol exceptions.

Instead of having to explicitly inject protocol exceptions using a directed approach, you can include these exceptions randomly.

Random injection of a protocol exception is accomplished by randomly generating an exception descriptor. This exception descriptor is implemented and generated using the same technique as transaction descriptors.

Example 6-14 shows an exception descriptor for an MII MAC-layer transactor that you can use to create collisions.

*Example 6-14 Exception Descriptor for an MII Protocol*

```
class mii_mac_collision;
   typedef enum {NONE, EARLY, LATE} kind_e;
   rand kind_e       kind;
   rand int unsigned on_symbol;
        int unsigned n_symbols;

   constraint early_collision {
      if (kind == EARLY) on_symbol < 112;
   }
   constraint late_collision {
      if (kind == LATE) {
         on_symbol >= 112;
         on_symbol < n_symbols;
      }
   }
   constraint no_collision {
      kind == NONE;
   }
endclass: mii_mac_collision
```

If more than one exception is injected concurrently during the execution of the transaction, the exception descriptor should properly model this capability.

The exception descriptor should contain a reference to the interacting transaction descriptor as shown in Example 6-15. This reference allows the expression of constraints to correlate protocol exceptions with the transactions they are applied to.

*Example 6-15 Exception Descriptor for an MII Protocol*

```
class mii_mac_collision;
   ...
   eth_frame frame;
   ...
endclass: mii_mac_collisions
```

To prevent the injection of protocol exception, an exception descriptor must be able to describe a no-exceptions condition as shown in Example 6-16. A constraint block should ensure that. By default, no exceptions are injected.

Most of the testcases show no interest in exceptions and thus use the transactor as-is. For the few tests responsible for verifying the response of the design to protocol exception, they simply need to turn off the constraint block.

*Example 6-16 Enabling the Injection of Protocol Exceptions*

```
class test_collisions extends vmm_test;
...
virtual function start_of_test_ph;
   env.phy.randomized_col.
      no_collision.constraint_mode(0);
endfunction
endclass: test_collisions
```

The random exception generation might be built in the transactor itself, as shown in Example 6-17. However, this usage requires that the author of the transactor plan for every possible exception that he can inject. If the source code for the transactor is available, the kinds of exceptions that the transactor can inject evolve according to the needs of the projects. You should never modify a truly reusable transactor.

If the source code is not available, it might be difficult to introduce additional exceptions in the transactor without introducing disjoint control mechanisms.

*Example 6-17 Exception Generation Built Into a Transactor*

```
class mii_phy_layer extends vmm_xactor;
   virtual mii_if.phy_layer sigs;
   ...
   mii_phy_collision randomized_col;

   function new;
      ...
      this.randomized_col = new;
   endfunction: new
   ...
   task tx_driver();
      ...
      if (!randomized_col.randomize()) ...
      ...
   endtask: tx_driver
endclass: mii_phy_layer
```

You can also build the random exception generation into a callback extension. You can use this mechanism to add exception injection capabilities into a transactor that does not already support them or to supplement the exceptions the transactor already provides.

Example 6-18 shows how you build the exception generation into a callback extension.

*Example 6-18 Exception Generation in a Callback Extension*

```
class gen_rx_errs extends mii_phy_layer_callbacks;
   mii_rx_err randomized_rx_err;
   ...
   virtual task pre_frame_tx(...);
      ...
      if (!randomized_rx_err.randomize()) ...
   endtask: pre_frame_tx

   virtual task pre_symbol_tx(...);
     if (this.randomized_rx_err.on_symbol == nibble_no)
```

```
            err = 1'b1;
        endtask: pre_symbol_tx
    endclass: gen_rx_errs
```

## Embedded Stimulus

Stimulus is generally understood as being applied to the external inputs of the design under verification. However, limiting stimulus to external interfaces might only make it difficult for you to perform some testcases.

If the verification environment does not have a sufficient degree of controllability over the design, you might spend much effort trying to create a specific stimulus sequence to an internal design structure. This is because; it is too far removed from the external interfaces. This problem is particularly evident in systems where internal buses or functional units are not directly controllable from the outside.

You might not need transactors to be limited to driving external interfaces. You can use them to replace an internal design unit and provide control over that unit's interfaces.

The transaction-level interface of the embedded transactor remains externally accessible, making the replaced unit interfaces logically external. You can similarly replace monitors for slave devices.

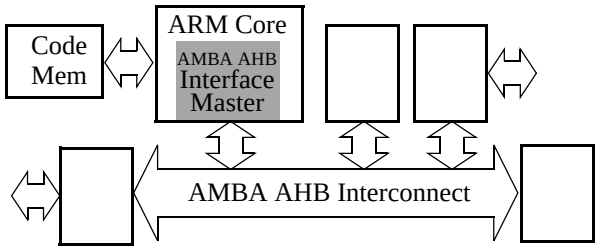
For example, an embedded ARM core can be replaced with an AMBA AHB Interface master transactor as shown in Figure 6-2.

Example 6-19 and Example 6-20 show how to instantiate an interface in a module and to bind it in the top environment.

Thus generation if transactions is achieved not by executing instructions but by having the transactor execute the transaction descriptors.

Connectivity is preserved and verified it because the transactor is inserted within the original design unit interface. The testcase run-time is improved, because fewer lines of code are simulated. There is no need to fetch instructions or the processor core executes no object code.

*Figure 6-2 Replacing a Design Unit With a Transactor*




*Example 6-19 Replacement Module for Embedded Stimulus Generation*

```
module arm_core(input  hclk,
                output mhbusreq,
                input  mhgrant, ...);

ahb_if sigs();
assign sigs.hclk    = hclk;
assign mhbusreq     = sigs.mhbusreq;
assign sigs.mhgrant = mhgrant;
...
endmodule
```

*Example 6-20 Embedded Transactor (Explicitly Phased Environment)*

```
task dut_env::build();
   ahb_master core = new(...);
   //bind the transactor virtual interface
   core.bind_vif(top.dut.core_i.sigs.master);
   ...
endtask
```

When substituting a design block for a transactor, you might need to ensure that the generated stimulus is representative of system-level activity.

## Controlling Random Generation

The objective of a random generator is to create the entire needed stimulus to completely verify a design. Some of these stimulus are created without any constraints, except those are required to create valid stimulus.

Other stimulus requires additional or modified constraints to strike certain corner cases or inject errors.

The ability to create certain stimulus patterns is directly related to the ability to express the constraints that causes the patterns to be generated.

If it is not possible to express a constraint between two variables, it is not possible to create a relationship between their respective values. Declarative constraints can only be expressed across properties (or sub-properties) of a class, procedural constraints are expressed across disjoint variables using the `std::randomize with` statement.

You should prefer declarative constraints as they can be added, modified or removed without modifying or duplicating the generation code.

Instead of coding a directed testcase to verify a particular function of the design, it might be simpler to modify the constraints on the generators to increase the likelihood that they generate the required data streams on their own.

Because generators are always randomizing the same instance, it is possible to “remove” the *rand* mode on arbitrary properties - which for a particular test - must remain constant.

You might turn off the *rand* mode of some properties by default to prevent generation of invalid data. Errors can be injected by turning them back on and adding relevant constraints. This procedural constraint modification can be executed at any time during the execution of a testcase.

*Example 6-21 Controlling the rand Mode of a Class Property*

```
vmm_xactor host_src;
$cast(host_src,
  vmm_object::find_object_by_name("host_src"));
host_src.randomized_obj.dst = this.cfg.mac.addr;
host_src.randomized_obj.dst.rand_mode(0);
host_src.randomized_obj.src = this.cfg.dut_addr;
host_src.randomized_obj.src.rand_mode(0);
```

Because generators are always randomizing the same instance, it is possible to turn constraint blocks ON or OFF using the `constraint_mode()` method. This method can disable constraint blocks that might prevent the injection of errors or modify the distribution of the generated values and obtain a different distribution. This method can be executed as a procedural constraint modification at any time during the execution of a testcase.

*Example 6-22 Controlling Constraint Blocks*

```
class test_collisions extends vmm_test;
...
 virtual function start_of_test_ph;
   begin
    vmm_xactor phy;
    $cast(phy,
      vmm_object::find_object_by_name("host_phy"));
    phy.randomized_col.no_collision.constraint_mode(0);
   end
 endfunction
endclass: test_collisions
```

If the definition of a randomized *class* contains *extern constraint* blocks, you can define them for each testcase. This style requires the pre-existence of an undefined *extern constraint* block and you can use it to add constraints.

The new *constraint* block definition can be simply added by including a source file that defines it. This change is a declarative constraint modification that applies to all instances of the *class*. They are taken into consideration (unless the *constraint* block is turned OFF) whenever you randomize an instance of the *class*. The constraints apply for the entire duration of the testcase execution.

*Example 6-23 Specifying External Constraints*

```
class test extends vmm_test;
...
  constraint eth_frame::tc1 {
    data.size() == min_len;
  }
endclass: test
```

It is not always possible to create the desired data stream simply by turning constraints on or off or by tweaking distribution weights. If the constraints or variable distribution weights did not exist earlier, it is not possible to create the necessary stimulus.

Because generators are always randomizing the same instance, it is possible to replace the randomized instance with an instance of a derived class using the factory service.

As the *randomize()* method is virtual, the additional or overridden constraint blocks should be implemented in the derived class.

Unlike the external constraint block implementation, this mechanism allows the addition of class properties and methods. It allows further extension of virtual methods to facilitate the expression of the required constraints. It also allows the redefinition of existing constraint blocks and methods.

Though the class extension is declarative and global to a simulation, the substitution of the randomized instance with an instance of this new class is procedural. This constraint modification can be done at any time during the execution of a testcase.

*Example 6-24 Replacing a Factory Instance*

```
class test;
...
class long_eth_frame extends eth_frame;
   `vmm_typename(long_eth_frame)
    constraint long_frames {
      data.size() == max_len;
   }
endclass: long_eth_frame
...
virtual function start_of_test_ph;
   begin
     //override default with long_eth_frame derived type
      eth_frame::override_with_new(
        "@env:host_src:randomized_obj",
        long_eth_frame::this_type,
        log);
   end
endfunction
endclass: test
```

*Example 6-25 Constraining the Test Configuration*

```
class duplex_test_cfg extends test_configuration;
   `vmm_typename(duplex_test_cfg)
   constraint test_Y {
      mode == DUPLEX;
   }
endclass

class test_Y;
```

```
    virtual function start_of_test_ph;
      begin
        test_configuration::override_with_new(
          "@top.env.randomized_cfg",
          duplex_test_cfg::this_type,
          log);
      end
    endfunction
  endclass: test_Y
```

## Modeling Scenarios

The atomic generator creates a stream of individually randomized transactions. This is fine for creating broad-spectrum stimulus, but corner cases are likely to require a more constrained sequence of transactions.

Scenarios are short sequences of transactions that are directed or mutually constrained, or a combination of both.

This chapter describes specification of single-stream and multi-stream scenarios - both random and directed - and hierarchical scenarios.

Note: The multi-stream scenarios are the recommended way to model scenarios going forward.

Appendix A includes detailed documentation for, `vmm_scenario_gen` and `vmm_scenario::define_scenario()`, `vmm_ms_scenario_gen` and `vmm_ms_scenario`.

## Architecture of the Generators

The scenario generators and multi-stream scenario generators are transactors that repeatedly select a scenario from a set of available ones. They randomize and then execute it. After you have executed a scenario, the total number of transactions the scenario creates is added to the total number of transactions the generator generates.The number of generated scenarios is incremented.

This process is repeated until the maximum number of scenarios or transaction descriptors to generate is reached.

By default, the single-stream scenario generator provides only one scenario: a scenario that randomizes and then applies just one transaction.

Functionally, the default behavior of the single-stream scenario generator is equivalent to that of the atomic generator.

You must register single-stream scenarios with a single-stream scenario generator to produce different stimulus. Note that the performance of the default-configuration single-stream scenario generator is significantly lower than the atomic generator because of the overhead associated with selecting, randomizing and applying scenarios.

You should not use this as a replacement of the atomic generator in situations where the atomic generator suffices.

By default, the multi-stream scenario generator does not provide any scenarios. Multi-stream scenarios must be registered with a multi-stream scenario generator to produce stimulus.

Other than this difference, multi-stream scenarios provide a more flexible feature set and you can use them in conjunction with existing single stream scenarios. It is therefore recommended over single stream scenario generator.

You can register scenarios to the desired scenario generator instance via the `vmm_scenario_gen::register_scenario()` or `vmm_ms_scenario_gen::register_ms_scenario()` method.

This allows specific generators to generate the desired stimulus sequence and no other. In case you need to register a scenario with multiple instances of scenario generators, you can use the transactor iterator as shown in Example 6-26.

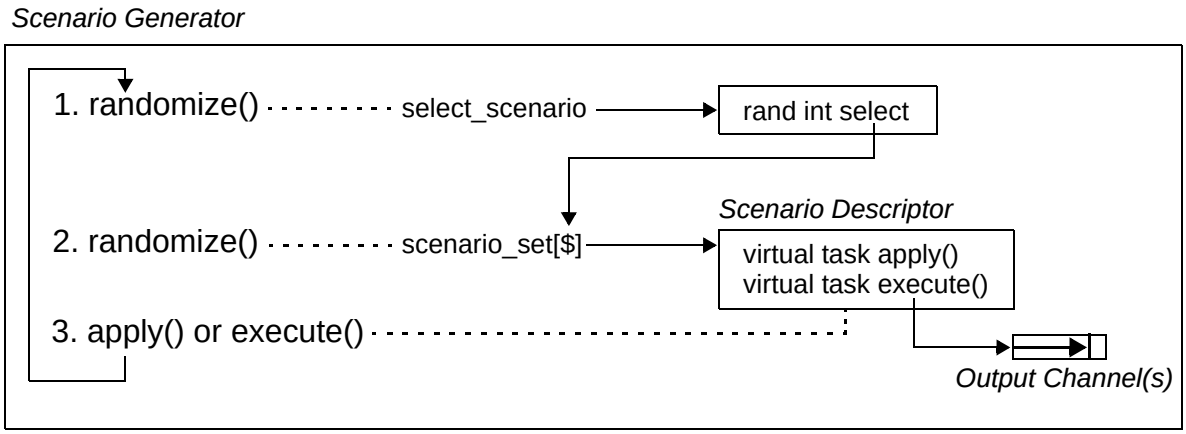
*Example 6-26 Registering a Scenario With Multiple Generators*

```
`foreach_vmm_xactor(ahb_scenario_gen, "/./", "/./") begin
   my_ahb_scenario sc = new();
   xact.register_scenario(sc);
end
```

## Scenario Selection

As shown in Figure 6-3, a generator selects to generate the next scenario among all of the scenarios you register with it, by randomizing its `vmm_scenario_gen::select_scenario` or `vmm_ms_scenario_gen::select_scenario` class property. The final value of the `vmm_scenario_election::select` or `vmm_ms_scenario_election::select` identifies the scenario. The generator interprets it as the index in the `vmm_scenario_gen::scenario_set[$]` or `vmm_ms_scenario_gen::scenario_set[$]` of the scenario generated.

*Figure 6-3 Scenario Selection and Execution Process*




By default, the `vmm_scenario_election::round_robin` and `vmm_ms_scenario_election::round_robin` constraint blocks constrains the selection process to a round-robin order.

By turning off this constraint block, you can make the scenario selection process completely random.

*Example 6-27 Making the Scenario Selection Random*

```
vmm_xactor gen;
$cast(gen,
  vmm_object::find_object_by_name("@env:ahb_gen2"));
gen.select.round_robin.constraint_mode(0);
```

You can replace the instance of the `vmm_scenario_election` or `vmm_ms_scenario_election` `class` in the `vmm_scenario_gen::select_scenario` or `vmm_ms_scenario_gen::select_scenario` class property to create a different selection process. Various state variables are available to help procedurally or randomly determine the next scenario to execute.

# Modeling Generators

## Atomic Generation

*Atomic generation* is the generation of individual data items or transaction descriptors. It generates each of them independent of the items or descriptors that was previously or subsequently generated.

Atomic generation is like using a random function that returns a complex data structure instead of a scalar value.

Atomic generation is simple to describe and use as shown in Example 6-28. Its ease of use is the reason why you use atomic generation to illustrate most of the generation and constraints examples in this book and in other literature.

However, it is unlikely to create interesting stimulus sequences on its own even with the addition of constraints.

*Example 6-28 Atomic Generator*

```
class eth_frame_gen extends vmm_xactor;
   ...
   eth_frame randomized_fr;
   ...
   virtual protected task main();
      ...
      while (...) begin
         ...
         if (!this.randomized_fr.randomize()) ...
         ...
      end
      ...
   endtask: main
endclass: eth_frame_gen
```

For example, how can you constrain an atomic instruction generator to generate a well-formed loop structure? How about a nested loop structure? Generating interesting stimulus sequences requires the ability to constrain random stimulus within the context of the previous and subsequent items and descriptors.

The predefined atomic generator *vmm_atomic_gen* macro creates follows all relevant guidelines. You can create with a few keystrokes, a powerful atomic generator for any type derived from the *vmm_data* class.

## Multiple-Stream Scenarios

Multi-stream scenarios are able to inject stimulus on multiple output channels. Unlike single-stream scenarios, you do not tie multi-stream scenarios to a particular channel. They have the flexibility to access any channel in the environment. You must explicitly define them by extending their `vmm_ms_scenario::execute()` method.

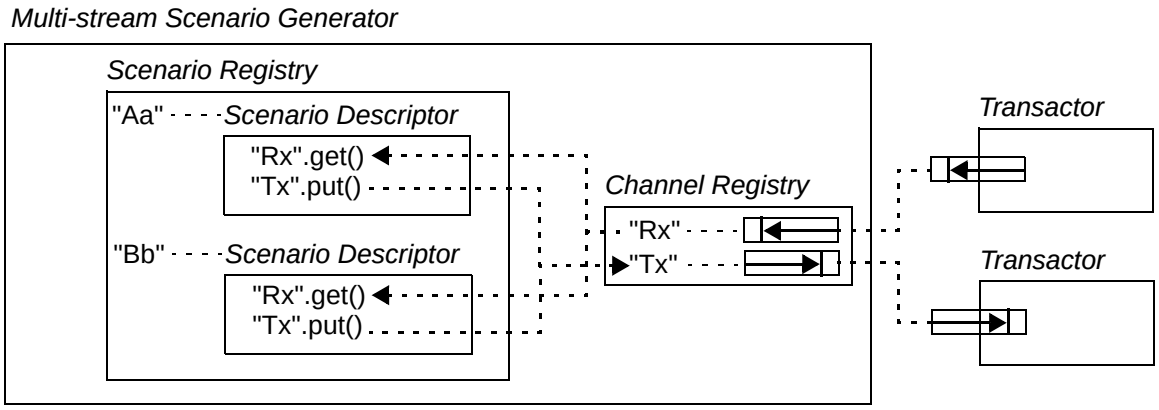
That is not to say that random multi-stream scenarios are not possible! You can implement a random multi-stream scenario by defining properties as “rand” or by calling “randomize()” from within the `vmm_ms_scenario::execute()` method.

As shown in Figure 6-4, multi-stream scenarios interact with channels identified by logical names. This allows to execute the same scenario on a different set of channels.

Channels are associated with a logical name by registering them with an instance of a multi-stream scenario generator by using the `vmm_ms_scenario_gen::register_channel()` method.

The channel that is associated with a logical name can be obtained from within the `vmm_ms_scenario::execute()` method by calling the `vmm_ms_scenario::get_channel()` method.

*Figure 6-4 Channels in Multi-Stream Scenarios*




*Example 6-29 Registering Logical Channel Pairs*

```
foreach (this.ms_gen[i]) begin
   this.ms_gen[i].register_channel("Tx",
                                   this.bfm[i].tx_chan);
   this.ms_gen[i].register_channel("Rx",
                                   this.bfm[i].rx_chan);
end
```

A multi-stream scenario need not only to generate stimulus on multiple output channels. You can use a single-channel multi-stream scenario to describe a single-stream scenario.

Similarly, you might connect a multi-stream scenario generator to only one output channel thereby effectively emulating a single-stream scenario generator.

The performance of a multi-stream scenario generator used in a single-stream application is comparable to the performance of a single-stream scenario generator.

## Procedural Scenarios

Multi-stream scenarios are procedural scenarios, which do not have a pre-defined default random scenario. The only implicit randomization is the randomization of the multi-stream scenario descriptor before you execute it.

The body of the multi-stream scenario is completely under your control and could include further randomization of local variables and data members. Or the hierarchical execution of child scenarios, depending on the your intention.

You must specify a multi-stream scenario by overloading the `vmm_ms_scenario::execute()` task in an extension of the `vmm_ms_scenario` class. You must specify each multi-stream scenario as a separate class extension. The execution of this task constitutes the multi-stream scenario.

It is required that for each scenario the `vmm_ms_scenario::copy()` should be overloaded for multistream scenarios to return the copy of the scenario.

The easiest way to achieve this is to use the shorthand macros.

```
`vmm_scenario_member_begin(..)
...
vmm_scenario_member_end(..)
```

Note: These macros create a default constructor. If there is a need to create your own constructor, you need to explicitly define the macro, `'vmm_scenario_new(..)` in addition to the above macros.

It is up to the task to create or randomize transaction descriptors and then copy them in the appropriate channels. It is recommended that,

- The transaction descriptor be factory enabled.
- That the scenario be factory enabled. This facilitates further customization or replacement of the scenario from a testcase.

*Example 6-30 A Simple Multi-Stream Scenario*

```
class simple_scenario extends vmm_ms_scenario;
   `vmm_typename(simple_scenario)
   rand ahb_cycle ahb;
        ocp_cycle ocp;

   function new(vmm_ms_scenario parent = null);
      super.new(parent);
      this.ahb =
        ahb_cycle::create_instance(this,"ahb_cycle");
      this.ocp =
        ocp_cycle::create_instance(this,"ocp_cycle");
   endfunction

   virtual function vmm_data copy(vmm_data to = null);
      simple_scenario cpy;

      if (to == null)
         cpy = new(this.get_parent_scenario());
      else $cast(cpy, to);

      $cast(cpy.ahb, this.ahb.copy());
      $cast(cpy.ocp, this.ocp.copy());
   endfunction

   virtual task execute(ref int n);
      vmm_channel ocp_chan = this.get_channel("OCP");
      vmm_channel ahb_chan = this.get_channel("AHB");
      fork
         begin
            this.ocp.randomize();
            ocp_chan.put(this.ocp.copy());
         end
         // this.ahb will be randomized when this
         // class is randomized by the generator
         ahb_chan.put(this.ahb.copy());
      join
      n += 2;
   endtask
   `vmm_class_factory(simple_scenario)
```

```
endclass
```

A multi-stream scenario generator can be connected to any channel instance in the testbench environment. However, such a connection does not prevent other transactors to concurrently inject transactions to a channel, as a scenario is not guaranteed exclusive access by default to an output channel.

Multiple threads in the same scenario might inject transactions in the same channel, or another generator might be actively generating its own stream of transactions in a channel concurrently with the multi-stream generator.

If a multi-stream scenario requires exclusive access to a channel, to ensure that you do not interrupt its specific sequence of transactions or mix with a sequence from another thread in the same scenario (or from another transactor) it must first grab the channel. This is done by calling the `vmm_channel::grab()` method.

After the channel is grabbed, all other potential producers on the channel are blocked from injecting transactions in the channel until it has been explicitly ungrabbed.

When injecting transactions in a potentially grabbed channel, you must supply a reference to the scenario currently injecting the transaction to grabber argument of the `vmm_channel::put()` or `vmm_channel::sneak()` methods.

*Example 6-31 A Multi-Stream Scenario With Exclusive Channel Access*

```
class exclusive_access extends vmm_ms_scenario;
   `vmm_typename(exclusive_access)
   rand ahb_cycle ahb;

   function new(vmm_scenario parent = null);
      super.new(parent);
      this.ahb = ahb_cycle::create_instance(this,"ahb_c");
```

```
   endfunction

   virtual function vmm_data copy(vmm_data to = null);
      exclusive_scenario cpy;

      if (to == null)
         cpy = new(this.get_parent_scenario());
      else $cast(cpy, to);

      $cast(cpy.ahb, this.ahb.copy());
   endfunction

   virtual task execute(ref int n);
      vmm_channel chan = this.get_channel("AHB");
      chan.grab(this);
      repeat (10) chan.put(this.ahb, .grabber(this));
      chan.ungrab(this);
      n += 2;
   endtask
`vmm_class_factory(exclusive_access)
endclass
```

## Hierarchical Scenarios

Multi-stream scenarios can be composed of other single-stream and multi-stream scenarios. There are two types of hierarchical scenarios: "***contained***" and "***distributed***".

A ***contained*** multi-stream scenario is entirely described and executed by a multi-stream scenario descriptor. It executes within the context of a single multi-stream scenario generator, as shown in Figure 6-4. The sub-scenarios in a contained hierarchical scenario execute on the same logical channels as the top-level scenario.

*Example 6-32 Contained Hierarchical Multi-Stream Scenario*

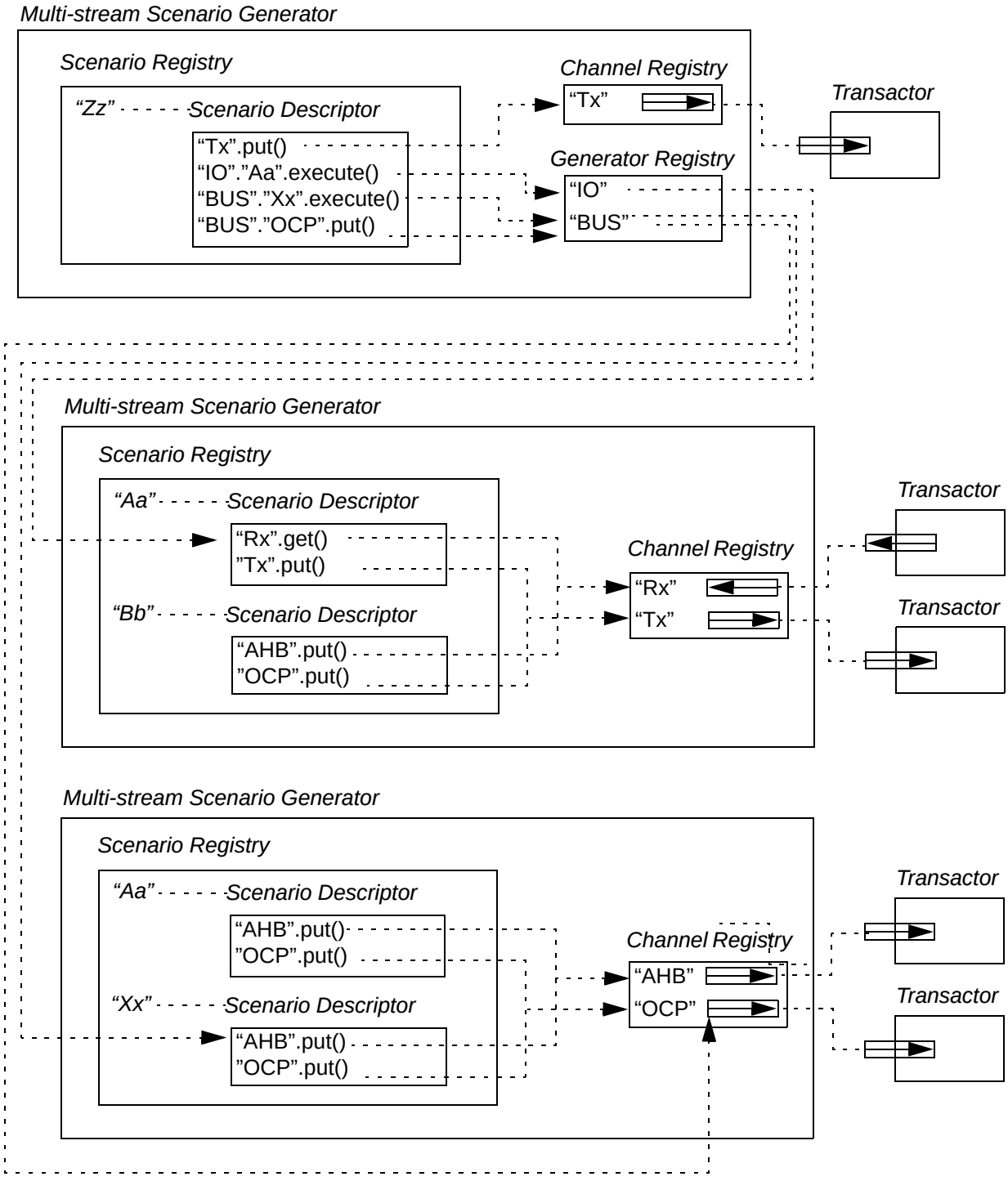
```
class contained extends vmm_ms_scenario;
   rand simple_scenario        simple;
   rand exclusive_access       excl;
   rand single_stream_scenario sss;
```

```
   function new(vmm_scenario parent = null);
      super.new(parent);
      this.simple = simple_scenario::create_instance(...);
      this.excl  = exclusive_access::create_instance(...);
      this.sss   =
         single_stream_scenario::create_instance(...);
      this.sss.set_parent_scenario(this);
   endfunction

   virtual function vmm_data copy(vmm_data to = null);
      contained cpy;

      if (to == null)
         cpy = new(this.get_parent_scenario());
      else $cast(cpy, to);

      $cast(cpy.simple, this.simple.copy());
      $cast(cpy.excl,   this.excl.copy());
      $cast(cpy.sss,    this.sss.copy());
   endfunction

   virtual task execute(ref int n);
      fork
         begin
            this.simple.execute(n);
            this.excl.execute(n);
         end
         this.sss.apply(this.get_channel("MII"), n);
      join
   endtask
endclass
```

A ***distributed*** multi-stream scenario is described and executed by multiple multi-stream scenario descriptors. Each multi-stream scenario descriptor executes within the context of the multi-stream scenario generator where it is registered, as shown in Figure 6-5.

The sub-scenarios in a distributed hierarchical scenario execute on the logical channels as registered in the multi-stream scenario generator where they execute.

*Figure 6-5 Distributed Hierarchical Multi-Stream Scenarios*



*Example 6-33 Distributed Hierarchical Multi-Stream Scenario*

```
class distributed extends vmm_ms_scenario;
   rand simple_scenario        simple;

   function new(vmm_scenario parent = null);
      super.new(parent);
      this.simple = simple_scenario::create_instance(...);
      this.simple.set_parent_scenario(this);
   endfunction

   virtual function vmm_data copy(vmm_data to = null);
      contained cpy;

      if (to == null)
         cpy = new(this.get_parent_scenario());
      else $cast(cpy, to);

      $cast(cpy.simple, this.simple.copy());
   endfunction

   virtual task execute(ref int n);
      fork
         this.simple.execute(n);
         begin
            vmm_ms_scenario mii;
            mii = this.get_ms_scenario("MII_GEN",
                                       "Collision");
            if (mii != null) mii.execute(n);
         end
      join
   endtask
endclass
...
initial
begin
   vmm_ms_scenario_gen top_gen = new;

   // Assuming that somewhere, this registration happens
   // env.mii_gen.register_ms_scenario("Collision", ...);

   top_gen.register_ms_scenario_gen("MII_GEN",
                                    env.mii_gen);
   begin
      distributed d = new;
      top_gen.register_ms_scenario("Example", d);
   end
```

```
   ...
end
```

You can surely compose a distributed hierarchical scenario of contained hierarchical scenarios.

## Configuring Scenario Generators

Scenario generators are configured by using many concurrent mechanisms. You can use various mechanisms or in combination with others to achieve the desired results.

### Stopping a Generator

The `stop_after_n_scenarios` class property specifies the total number of scenarios to generate. By default, it is set to zero or an infinite number of scenarios.

Example 6-34 shows how to configure a specific scenario generator instance to automatically stop after generating one scenario.

*Example 6-34 Configuring the Number of Scenarios to Generate (Explicit Phasing)*

```
task my_test::run(vmm_env env);
  vmm_xactor gen0;
   $cast(gen0, vmm_object::find_object_by_name("eth_gen"));
  env.build();
  gen0.stop_after_n_scenarios = 1;
  env.run();
endtask
```

The `stop_after_n_insts` class property specifies the minimum total number of transactions to generate. By default, it is set to zero or an infinite number of transactions.

Example 6-35 shows how to configure all instances of a scenario generator type to automatically stop after generating at least one hundred transactions.

*Example 6-35 Configuring the Number of Transactions to Generate (Explicit Phasing)*

```
task my_test::run(vmm_env env);
   env.build();
   begin
      `foreach_vmm_xactor(ahb_scenario_gen, "/./", "/./")
         xact.stop_after_n_insts = 100;
   end
   env.run();
endtask
```

### Available Scenarios

The scenarios that are available to be generated by the generator must be registered with a generator. By default, single-stream scenario generators only know about the "atomic" scenario and multi-stream scenario generators do not know about any scenarios.

Example 6-36 shows how to remove the default atomic scenario from a single-stream scenario generator instance.

*Example 6-36 Removing Scenarios (Explicit Phasing)*

```
task my_test::run(vmm_env env);
   vmm_xactor gen;
   $cast(gen,
      vmm_object::find_object_by_name("atomic_gen"));
   env.build();
   gen.unregister_scenario_by_name("Atomic");
   env.run();
endtask
```

Example 6-37 shows how to register a user-defined scenario with all instances of a specific scenario generator class.

*Example 6-37 Registering Scenarios*

```
class a_scenario extends
   ahb_scenario;
   ...
endclass

task my_test::run(vmm_env env);
   env.build();
   begin
      `foreach_vmm_xactor(ahb_scenario_gen,
                          "/./", "/./") begin
         a_scenario a = new;
         xact.register_scenario("A", a);
      end
   end
   env.run();
endtask
```

It is also possible to design a verification environment where scenarios can be automatically registered with all instances of the relevant scenario generators.

Example 6-38 shows how to implement a type-specific global scenario registry and automatic scenario registration in an environment.

*Example 6-38 Automatic Scenario Registration*

```
class auto_ahb_scenario extends ahb_scenario;
   static string names[$];
   static ahb_scenario registry[$];
   static function bit auto_register(string name,
                                     ahb_scenario sc);
      this.names.push_back(name);
      this.registry.push_back(sc);
   endfunction
endclass

class a_scenario extends auto_ahb_scenario;
   ...
   static local a_scenario _sc = new();
   static local bit _dummy = auto_register("A", this._sc);
endclass
```

```
class b_scenario extends auto_ahb_scenario;
   ...
   static local b_scenario _sc = new();
   static local bit _dummy = auto_register("B", this._sc);
endclass

class tb_env extends vmm_env;
   ...
   virtual task build();
      ...
      foreach (auto_ahb_scenario::names[i]) begin
         `foreach_vmm_xactor(ahb_scenario_gen,
                             "/./", "/./")
            xact.register_scenario(
            auto_ahb_scenario::names[i],
            auto_ahb_scenario::registry[i]);
      end
   endtask
   ...
endclass
```

### Scenario Generation Order

The next scenario to generate is defined by randomizing their respective *select_scenario* class property. By default, scenarios are selected in a round-robin fashion. You can modify the scenario selection by changing the constraints on the *select* subclass property.

Example 6-39 shows how to configure the scenario selection process for all multi-stream scenario generator instances to select one specific scenario, another specific scenario and finally randomly make a selection from the remaining scenarios.

*Example 6-39 Configuring the Scenario Selection Process (Explicit Phasing)*

```
class a_then_b_then_random extends
   vmm_ms_scenario_election;
   constraint round_robin {
```

```
      if (scenario_id == 0) select == 0;
      if (scenario_id == 1) select == 1;
      if (scenario_id > 1) select > 1;
   }
endclass

task my_test::run(vmm_env env);
   env.build();
   begin
      a_then_b_then_random sel = new;
      `foreach_vmm_xactor(vmm_ms_scenario_gen,
                          "/./", "/./") begin
         a_scenario a = new;
         b_scenario b = new;
         xact.scenario_set.push_front(b);
         xact.scenario_set.push_front(a);
         xact.select_scenario = sel;
      end
   end
   env.run();
endtask
```

You might implement a "directed" testcase by running one "top-level" scenario. You can accomplish this by making sure this top-level scenario is the first one selected by pushing it at the front of the `scenario_set` array and configuring the generator to execute only one scenario.

Example 6-40 assumes the existence of a top-level multi-stream scenario generator to execute such a directed testcase.

*Example 6-40 Running Only One Top-Level Scenario (Explicit Phasing)*

```
class directed_test extends
   vmm_ms_scenario;
   ...
endclass

task my_test::run(vmm_env env);
   env.build();
   begin
      vmm_xactor top_gen;
      $cast(top_gen,
```

```
      vmm_object::find_object_by_name("top_gen"));
    directed_test test = new;
    top_gen.push_front(test);
    top_gen.stop_after_n_scenarios = 1;
  end
  env.run();
endtask
```

### Constraining Transactions

The *items* class property in single-stream scenario descriptors implements a two-stage factory for generating random transactions:

- The array is filled with copies of the using class property, if it is not null. Once filled, the array is repeatedly randomized and then its result content is copied onto the output channel.
- To modify the constraints on all the transactions in a single-stream scenario descriptor, you might assign a prototype instance to the *using* class property, as shown in Example 6-41.

  The alternative and recommended technique is to use VMM class factory service. For details see "Factory for Scenario Generators" on page 38.

Note: It is important that you properly overload the `copy()` method in the transaction class extension. This will ensure that the *items* array is filled with instances of the using class property.

*Example 6-41 Modifying Constraints in All Transactions (Explicit Phasing)*

```
class my_ahb_tr extends ahb_tr;
  constraint my_constraints {
    ...
  }
  `vmm_data_member_begin(my_ahb_tr)
  `vmm_data_member_end(my_ahb_tr)
endclass

task my_test::run(vmm_env env);
```

```
   env.build();
   begin
      my_ahb_tr tr = new;
      foreach (env.gen.scenario_set[i]) begin
         env.gen.scenario_set[i].using = tr;
      end
   end
   env.run();
end
```

To modify the constraints on a specific transaction in a single-stream scenario descriptor, you should assign the specialized instance to the required *items* element as shown in Example 6-42. The remaining array elements are filled with default instances.

*Example 6-42 Modifying Constraints in a Specific Transaction*

```
class my_ahb_tr extends ahb_tr;
   constraint my_constraints {
      ...
   }
   `vmm_data_member_begin(my_ahb_tr)
   `vmm_data_member_end(my_ahb_tr)
endclass

task my_test::run(vmm_env env);
   env.build();
   begin
      ahb_tr_scenario sc;
      my_ahb_tr tr = new;
      vmm_xactor gen;
      $cast(gen,
         vmm_object::find_object_by_name("ahb_gen"));
      sc = gen.get_scenario("Aa");
      sc.items.fill_scenario();
      sc.items[0] = tr;
   end
   env.run();
endtask
```

To modify the constraints on the components of a multi-stream scenario descriptor or a hierarchical single-stream scenario descriptor, you might assign the prototype property in the scenario with the specialized, derived instance as shown in next example.

The alternative and recommended technique is to use the VMM class factory service as shown in “Factory for Scenario Generators” on page 38.

*Example 6-43 Modifying Constraints in Other Scenario Descriptors (Explicit Phasing)*

```
class my_ahb_tr extends ahb_tr;
   constraint my_constraints {
      ...
   }
   `vmm_data_member_begin(my_ahb_tr)
   `vmm_data_member_end(my_ahb_tr)
endclass

task my_test::run(vmm_env env);
   env.build();
   begin
      some_scenario sc;
      my_ahb_tr tr = new;
      sc = env.gen.get_scenario("Aa");
      sc.ahb = tr;
   end
   env.run();
endtask
```

## Single-Stream Scenarios

The single-stream scenario generator is a type-specific generator that you declare using the `vmm_scenario_gen()` macro as shown in Example 6-44. This creates a class named `class_name_scenario_gen` where `class_name` is the name of the user-defined class you supply to the macro.

Alternatively, you might also declare the scenario generator might using VMM built-in parametrized implementations of the generators which is described in “Parameterized Atomic and Scenario Generators” on page 52.

*Example 6-44 Declaring a Single-Stream Scenario Generator*

```
class eth_frame extends vmm_data;
   ...
endclass
`vmm_channel(eth_frame)
`vmm_scenario_gen(eth_frame, "Ethernet Frames")
```

You connect the single-stream scenario generator to a single output channel at construction time, or by assigning its `vmm_scenario_gen::out_chan` class property. All generated scenarios are injected in this output channel.

*Example 6-45 Instantiating a Single-Stream Scenario Generator (Explicit Phasing Environment)*

```
class tb_env extends vmm_env;
   eth_frame_scenario_gen gen;
   eth_frame_channel gen_to_bfm;
   ...
   virtual function void build();
      super.build();
      this.gen_to_bfm = new();
      this.gen = new("gen", 0, this.gen_to_bfm);
   endfunction
   ...
endclass
```

The macro also defines a single-stream scenario descriptor class named `class_name_scenario`. This class contains a type-specific array of transaction descriptors that you randomize according to the constraints in the scenario descriptor.

The macro predefines an atomic scenario in a class named `class_name_atomic_scenario`. You register an instance of this class by default with any instance of the corresponding single-stream scenario generator. You will need to unregister this default scenario if you do not desire it.

## Random Scenarios

By default, single-stream scenarios are randomly generated. The combination of three things makes this happen:

- A single-stream scenario descriptor contains a *`rand`* array of user-defined transaction descriptors in the *`class_name`*`_scenario::items[]` class property.
- After the scenario descriptor is selected, the generator automatically randomizes it.
- The default behavior of the *`class_name`*`_scenario::apply()` method copies the content of the *`class_name`*`_scenario::items[]` class property onto the generator's output channel.

As shown in Example 6-46, you define a random scenario by extending the `class_name_scenario` class and providing constraints over the elements of the `class_name_scenario::items[]` class property.

You must also specify the maximum length of the scenario by calling the `vmm_scenario::define_scenario()` method. This process is simplified by the use of the shorthand macros for scenario generators.

*Example 6-46 Declaring a Random Single-Stream Scenario*

```
class bad_eth_frames extends eth_frame_scenario;
```

```
   `vmm_typename(bad_eth_frames)
   function new();
      this.define_scenario("Bad Frames", 10);
   endfunction

   constraint bad_eth_frames_valid {
      foreach (this.items) {
         this.items[i].fcs != 0;
      }
   }
   `vmm_class_factory(bad_eth_frames)
endclass
```

## Procedural Scenarios

Procedural or directed scenarios are specified by overloading the *class_name*_scenario::apply() method. Any user-defined code can use the procedural scenario that puts transaction descriptors into the supplied output channel. The total number of procedurally generated transactions is then returned via the n_insts argument.

Note: It is important that you do not call super.apply(), else any transaction descriptor found in the class_name_scenario::items[] class property will also be injected into the output channel.

You can create random transactions by using rand class properties (such as, the predefined class_name_scenario::items[] class property), or by explicitly calling randomize() on local variables or non-random class properties.

*Example 6-47 Declaring a Procedural Single-Stream Scenario*

```
class collision extends eth_frame_scenario;
   `vmm_typename(collision)
   virtual mii_if sigs;

   function new();
```

```
      bit is_set;
      mii_if_wrapper if_wrapper;
      this.define_scenario("Collision", 1);
      $cast(if_wrapper,
        vmm_opts::get_object_obj(is_set, this,
"mii_if_wrapper"));
      this.sigs = if_wrapper.sigs;
   endfunction

   virtual task apply(eth_frame_channel channel,
                      ref int unsigned  n_insts);
      @ (posedge this.sigs.crs);
      channel.put(this.items[0]);
      n_insts++;
   endtask
`vmm_class_factory(collision)
endclass
```

If stimulus from another scenario must not interrupt the sequence of transactions the scenario generates (For details, see “Multiple-Stream Scenarios” on page 29), it might take an output channel for exclusive use until it is explicitly released.

If another scenario does not take the channel, the scenario generator reserves it immediately for the exclusive use of this scenario descriptor. If another scenario does take the channel, the generator suspends the execution of this scenario descriptor until the channel becomes available.

*Example 6-48 Ensuring a Transaction Order in a Single-Stream Scenario*

```
class dot_dot_dot extends eth_frame_scenario;
   `vmm_typename(dot_dot_dot)
   function new();
      this.define_scenario("Exclusive", 0);
   endfunction

   virtual task apply(eth_frame_channel channel,
                      ref int unsigned  n_insts);
      eth_frame fr;

      fr = new;
```

```
      fr.randomize() with {...};
      channel.grab(this);
      repeat (3) begin
         channel.put(fr.copy(), .grabber(this));
      end
      channel.ungrab(this);
      n_insts += 3;
   endtask
`vmm_class_factory(dot_dot_dot)
endclass
```

## Hierarchical Scenarios

You can describe scenarios hierarchically by composing them of lower-level scenarios. A hierarchical scenario is a procedural scenario. You simply instantiate the lower-level scenario descriptors in the higher-level scenario descriptor.

The higher-level scenario's *`apply()`* method calls the lower-level scenario's respective *`apply()`* method in the appropriate sequence.

*Example 6-49 Declaring a Hierarchical Single-Stream Scenario*

```
class bad_frames_then_collision extends eth_frame_scenario;
   `vmm_typename(bad_frames)
   rand bad_eth_frames bad;
   rand collision      col;

   function new();
      this.define_scenario("Bad+Collision", 0);
      this.bad =
         bad_eth_frames::create_instance(this,"bad");
      this.col = collision::create_instance(this,"col");
   endfunction

   virtual task apply(eth_frame_channel channel,
                      ref int unsigned  n_insts);
      this.bad.apply(channel, n_insts);
      this.col.apply(channel, n_insts);
   endtask
```

```
`vmm_class_factory(bad_frames)
endclass
```

You register hierarchical scenarios like any other scenarios. If the sub-scenarios are relevant top-level scenarios, you need to register them for them to become available for selection.

*Example 6-50 Registering Hierarchical and Flat Scenarios*

```
`foreach_vmm_xactor(eth_frame_scenario_gen,
                    "/./", "/./") begin
   mii_phy                    phy;
   if ($cast(phy, xact.out_chan.get_consumer())) begin
      bad_frames_then_collision btc =
        bad_frames_then_collision::create_instance(
          this,"bad_col");
      bad_eth_frames            bad =
        bad_eth_frames::create_instance(this,"bad");

      xact.register_scenario("Bad then Col", btc);
      xact.register_scenario("Bad Burst", bad);
   end
end
```

To prevent deadlock situations, a higher-level-scenario-taken channel is available for its lower-level scenarios. To make the exclusive use of an output channel from a higher-level scenario available to a lower-level scenario it is necessary to specify that the higher-level scenario instance is a parent of the lower-level scenario.

*Example 6-51 Preventing Deadlocks in Taking the Output Channel*

```
class bad_frames_then_collision extends eth_frame_scenario;
   `vmm_typename(bad_frames_then_collision)
   rand dot_dot_dot    ddd;
   rand bad_eth_frames bad;
   rand collision      col;

   function new();
      this.define_scenario("Bad+Collision", 0);
      this.ddd = dot_dot_dot::create_instance(...);
      this.bad = bad_eth_frames::create_instance(...);
```

```
      this.col = collision::create_instance(...);

      this.ddd.set_parent_scenario(this);
      this.bad.set_parent_scenario(this);
      this.col.set_parent_scenario(this);
   endfunction

   virtual task apply(eth_frame_channel channel,
                      ref int unsigned  n_insts);
      channel.grab(this);
      this.bad.apply(channel, n_insts);
      this.col.apply(channel, n_insts);
      channel.ungrab(this);
   endtask
`vmm_class_factory(bad_frames_then_collision)
endclass
```

## Parameterized Atomic and Scenario Generators

In addition to the macro-based definition of built-in VMM atomic and scenario generators, parameterized implementations are available. These also contain built-in class factories. For details, see “Factory for Atomic Generators” on page 36.

The main classes created for this purpose are,

- `class vmm_atomic_gen #(type T)`
- `class vmm_scenario_gen #(type T)`
- `class vmm_ss_scenario #(type T)`

These are generic classes with parameterized transaction types. you define `` `vmm_atomic_gen ``/`` `vmm_scenario_gen `` macros in VMM library to use these parameterized atomic/scenario generators and scenarios using `typedef` as shown in the following examples:

- `typedef vmm_atomic_gen#(T) T_atomic_gen;`

- `typedef vmm_scenario_gen#(T) T_scenario_gen;`
- `typedef vmm_ss_scenario#(T) T_scenario;`

This ensures that the existing macro-based atomic/scenario usage is fully supported without making any changes in user code.

There are two methods to declare `vmm_channel`, `vmm_atomic_gen`, `vmm_scenario_gen` objects:

- By using macros
- By using parameterized classes

The first method is to instantiate generators and channels, which you define using macros, as shown in Example 6-52. You can use callbacks and scenarios in the same way.

*Example 6-52 Using Macros for Declaring Atomic and Scenario Generators*

```
class ahb_trans extends vmm_data;
   rand bit [31:0] addr;
   rand bit [31:0] data;
endclass

`vmm_channel(ahb_trans)
`vmm_atomic_gen(ahb_trans, "AHB Atomic Gen")
`vmm_scenario_gen(ahb_trans, "AHB Scenario Gen")

ahb_trans_channel chan0 = new("ahb_trans_chan", "chan0");
ahb_trans_atomic_gen   gen0  = new("AhbGen0", 0, chan0);
ahb_trans_scenario_gen gen1  = new("AhbGen1", 0, chan0);

class user_callbacks0 extends
   ahb_trans_atomic_gen_callbacks;
endclass

class user_callbacks1 extends
   ahb_trans_scenario_gen_callbacks;
endclass
```

```
class user_scenario extends ahb_trans_scenario;
endclass
```

The second method is to create a user-defined type using *typedef* or directly instantiate the parameterized generator and channels, as shown in Example 6-53. You must use the parameterized `vmm_ss_scenario` class in case of a single stream scenario, as the base class for user-defined single stream scenarios.

*Example 6-53 Parameterized Atomic and Scenario Generators*

```
vmm_channel_typed#(ahb_trans) chan0 = new(
   "ahb_trans_chan", "chan0");
vmm_atomic_gen #(ahb_trans) gen0 = new("AhbGen0", 0, chan0);
vmm_scenario_gen #(ahb_trans) gen1 = new(
   "AhbGen1", 0, chan0);

class user_callbacks0 extends
   vmm_atomic_gen_callbacks#(ahb_trans);
endclass

class user_callbacks1 extends
   vmm_scenario_gen_callbacks#(ahb_trans);
endclass

class user_scenario extends
   vmm_ss_scenario#(ahb_trans);
endclass
```

## Implementing Testcases

The `vmm_test` base class must be used to implement test cases. For each testcase, you should create a new class that extends `vmm_test`. You must implement a testcase using the phasing mechanism of the environment for which it is written.

You can write testcases using an implicitly-phased or explicitly-phased top-level environment. For details, see “Understanding Implicit and Explicit Phasing” on page 31.

## Creating an Explicitly Phased Test

When writing a test for an explicitly phased environment (that is, based on `vmm_env`), the test procedure is implemented in the `vmm_test::run()` method. Shorthand macros are available to simply the creation of such tests, as shown in Example 6-54.

*Example 6-54 Declaring Test Using vmm_test* Macros

```
`vmm_test_begin(test, my_env, "Test")
   // Body of run() task here...
   this.env.start();
   this.env.run();
`vmm_test_end(test)
```

## Creating an Implicitly Phased Test

While writing a test for an implicitly phased environment (that you base on `vmm_group`), you implement the test procedure by extending the appropriate phase methods.

Any test-specific `vmm_group` component that is a child of the test object and part of the top-test timeline is automatically phased.

A simple default test is implemented as shown in Example 6-55.

*Example 6-55 Declaring Test Using vmm_test* Extension

```
class test1 extends vmm_test;
   `vmm_typename(test1)

   function new();
```

```
      super.new("test1");
   endfunction

endclass: test1
```

A typical test changes or adds a few constraints to existing transactions, introduces modifications etc.

Example 6-56 shows how to add constraints to a generator by inserting it back using the object factory.

*Example 6-56 Implementing Test Using vmm_test*

```
`include "vip_trans.sv"
class test2_trans extends vip_trans;
   `vmm_typename(test2_trans)
   constraint { … }
   `vmm_data_member_begin(test2_trans)
   `vmm_data_member_end(test2_trans)
endclass: test2_trans

class test2 extends vmm_test;
   function new();
      super.new("test2");
   endfunction

   virtual function void configure_test_ph();
     // Replace factory transaction with extended type
     vip_trans::override_with_new("@%*",
          test2_trans::this_type(), log);
   endfunction
endclass: test2
```

## Running Tests

An ***explicitly phased*** verification environment can simulate only one test per run. The test is run by calling `vmm_test_registry::run()` in a program thread. If a single test

class exists in the simulation, that is the test that is run by default. If multiple test classes exist, you must specify the name of the test to run using the `+vmm_test` option.

Example 6-57 shows how to register multiple tests and run them, this is the recommended way for explicitly phased environments.

*Example 6-57 Multiple Tests Registration*

```
`vmm_test_begin(test1, my_env, "Test1")
   ...
`vmm_test_end(test1)

`vmm_test_begin(test2, my_env, "Test2")
   ...
`vmm_test_end(test2)

program top;

initial
begin
   my_env env = new;
   vmm_test_registry::run(env);
end
endprogram
```

An ***implicitly phased*** verification environment can simulate multiple tests per run, one after another. The tests are run by calling `vmm_simulation::run_tests()` in a program thread.

If a single test class exists in the simulation, that is the test which is run by default. You must specify the name of the test(s) - if multiple test classes exist - using the `+vmm_test` option and using a plus-separated list of test names.

Example 6-58 shows how to construct multiple tests and run them, this is the recommended way for implicitly phased environments.

*Example 6-58 Multiple Tests Registration*

```
program top;
   initial
   begin
      my_env env = new("env");
      test1 t1 = new("test1");
      test2 t2 = new("test2");
      vmm_simulation::run_tests();
   end
endprogram
```

It is important to note that when serializing multiple tests, they might not behave the same way when they are running as standalone. This unless special care is taken to ensure that they start with a clean slate.

VMM phasing provides tests the capability to restore the initial state at the end of the test using the `"configure_test"` phase.

# 7

# Common Infrastructure and Services

This chapter contains the following sections:


- “Common Object”
- “Message Service”
- “Class Factory Service”
- “Options & Configurations Service”
- “Simple Match Patterns”

# Common Object

## Overview

The `vmm_object` is a virtual class that is used as the common base class for all VMM-related classes. It provides parent/child relationships for all VMM class instances. Additionally, it provides local, relative and absolute hierarchical naming. Combined with regular expressions, it makes it easy to locate all specific objects that match a given pattern in any hierarchy. This base class comes with a rich set of methods for assigning, querying, printing and traversing object hierarchies.

This section contains the following topics:


- "Setting Object Relationships"
- "Finding Objects"
- "Printing and Displaying Objects"
- "Object Traversing"
- "Namespaces"


## Setting Object Relationships

All classes that are based on the `vmm_object` base class have constructors that include a reference to the `parent` object and an object `name` as optional arguments, which are then passed to the `vmm_object` class constructor.

You can define the `vmm_object` members parent and name explicitly using the `vmm_object::set_parent_object()` and `vmm_object::set_object_name()` methods respectively.

When a parent object is specified, the new object is added to the list of children objects in the parent object by default. If no parent object is specified, the new object is a new root object. Thus, the parent-child relationship is created when any class extending from `vmm_object` is created.

Example 7-1 shows how to build up the parent-child association during construction.

*Example 7-1 Building Object and Associating Parent-Child Relationship*

```
class cfg extends vmm_object;
   function new(vmm_object parent = null, string name = "");
      super.new(parent,name);
   endfunction
endclass

class vip extends vmm_xactor;
   cfg c1;
   function new(string name = "", string inst = "",
                vmm_object parent = null);
      super.new(name,inst, parent);
      c1 = new(this, "CFG");
   endfunction
endclass

class env extends vmm_group;
   vip v1;
   function new(string name = "", string inst = "",
                vmm_object parent = null);
      super.new(name,inst, parent);
      v1 = new("VIP", "v1", this);
   endfunction
endclass
```

```
class root_class extends vmm_object;
    function new(vmm_object parent = null, string name = "");
                super.new(parent,name);
    endfunction
endclass

program test;
   initial begin
      root_class orphan;
      env e1 = new ("env","e1");
      orphan = new(,"orphan"); //No parent
  end
endprogram
```

As described in Example 7-1, instances `c1`, `v1` and the parent `e1` are passed through the constructor and thus the parent-child association between them is established. It is possible not to associate a parent to an object, in this case the object belongs to the root object.

Example 7-2 shows how to change relationships and the new objects hierarchy as in Example 7-3.

*Example 7-2 Object Hierarchy Tree*

```
[e1]
|--[v1]
|-----[c1]
[orphan]
```

Note: The parent-child association is typically done when constructing object. It is possible to change this relationship by using the `vmm_object::set_parent_object()` method.

*Example 7-3 Changing Parent-Child Relationship*

```
c1.set_parent_object(orphan);
```

```
[e1]
|--[v1]
```

```
[orphan]
|--[c1]
```

It is a good practice to name an object after the variable that contains the reference to the newly created object. This way, it makes it easier to correlate an object name with its location in the actual class hierarchy.

If you decide not to add `vmm_object` to the list of children objects, you can do it by setting the flag `disable_hier_insert` to 1 in the argument of the constructor.

Though a `vmm_object` instance can be referred through different SystemVerilog class-handle variables of different names, it only has one `name` an one `parent`. If an object location in the class hierarchy changes throughout the simulation, its name and parent remain the same.

Although it is possible to update the `parent` and `name` of an object to reflect its new location, it is recommended that you do not modify them. The main reason is that some name-based registries may depend on the name of an object to locate it and the identity of the parent object is useful for identifying the origin of an object.

There are methods to query the parent and the children objects of an object. You can use these methods dynamically for structural introspection.

In Example 7-4, `get_parent_object()` invoked from instance `e1.v1.c1` returns parent object of `c1`, which turns out to be `v1`.

*Example 7-4 Getting Handle to Parent Object*

```
vmm_object parent;
env e1 = new ("env","e1");
```

```
parent = e1.v1.c1.get_parent_object();
// parent of c1 is now v1
```

## Finding Objects

Given that different components directly or indirectly extend the `vmm_object` base class, you can use pre-defined methods to query hierarchical names, find objects and children by name, find the root of an object and so on. While invoking these functions, you can use the simple match patterns or complete regular expressions to define the search criteria.

The `get_object_hiername()` and `get_object_name()` methods return the full hierarchical name and local name of the object respectively. A hierarchical name is composed of series of colon-separated object names, usually starting from a root object through parent-child relationships.

The methods `find_child_by_name()` and `find_object_by_name()` find the named object as a hierarchical name relative to this object or absolute hierarchical name respectively in the specified namespace.

The `get_nth_root()` and the `get_nth_child()` methods return the $n^{th}$ root and the $n^{th}$ child respectively of the specified object.

Example 7-5 shows how to use the various methods to find and query objects.

*Example 7-5 Finding and Querying Objects*

```
my_class inst1 = new("inst1");
initial begin
   vmm_object obj, root;
```

```
obj = e1.find_child_by_name("c1");
// obj is now c1

obj = e1.get_nth_child(0);
// obj name is now "v1"

root = E::get_nth_root(1);
// root name is now "orphan"
```

## Printing and Displaying Objects

At any point in time, it is possible to view the complete `vmm_object` hierarchy of the testbench or the sub-hierarchy of any instance of a `vmm_object` using the `print_hierarchy()` method.

This method displays the `vmm_object` hierarchy as currently defined by the parent-child relationships and object names. It only prints the hierarchy correctly if the right parent-child relationship has been created before the invocation of the `print_hierarchy()` method.

*Example 7-6 Printing Hierarchy of Objects*

```
vmm_object::print_hierarchy(e1);
```

The above code produces the following output:

```
[e1]
|--[v1]
|  |--[c1]
```

## Object Traversing

The `vmm_object_iter` class traverses the hierarchy rooted at the specified object, looking for objects whose relative hierarchical name matches the specified name. Beginning at a specific object, it can traverse through the hierarchy via the `vmm_object_iter::first()` and `vmm_object_iter::last()` methods.

Continuing from the previous example, Example 7-7 shows how to traverse an object hierarchy.

*Example 7-7 Traversing Object Hierarchy*

```
Object_extension my_obj; //object_extension is a class
inherited from vmm_object
  vmm__object_iter my_iter = new( e1, pattern);
 `vmm_note(log, $psprintf("Match pattern: %s with root e1",
                           pattern));
  my_obj = my_iter.first();
  // my_obj is now "v1"

  my_obj = my_iter.next();
  // my_obj is now NULL
```

`` `foreach_vmm_object `` is a powerful macro to iterate over all objects of a specified type and name under a specified root.

Example 7-8 shows how to traverse an object's hierarchy using `` `foreach_vmm_object `` macro.

*Example 7-8 Traversing Object Hierarchy Using Macro*

```
`foreach_vmm_object(vmm_object, "@%*", e1) begin
   `vmm_note(log, {"Got:", obj.get_object_name()});
end
```

## Namespaces

VMM introduces the concept of namespace for object. The main purpose of namespace is to attach objects to a given space. This is particularly useful when a given lower-stream transactor must execute transactions from various upper-stream transactors like multi-stream scenario generator, RAL and other transactors.

Because each upper-stream transactor can tag its transaction to be executed with its namespace, it is easier to determine where this transaction is coming from by simply looking into its namespace.

For instance, all transactions that a RAL application initiates belong to its space event though signal-level transactors execute them. More explicitly, if you initiate an abstract call to register like `my_ral.write(IRQ_EN, 32'h01)`, the associated bus transaction like `AXI.WRITE(32'h1000, 32'h01)` becomes tagged with the RAL namespace and you can easily associate its source.

You can specify a namespace optionally at the beginning of a pattern using the namespace scope operator ::. A namespace might contain any character except a colon (:). If you do not specify a namespace, you use the object namespace. An error is issued if an unknown namespace is specified.

For example, looking for a leaf object named “X” in the “RAL” namespace would be specified as,

```
RAL::%:X
```

```
Namespace names starting with “VMM” are reserved.
```

# Message Service

This section contains the following topics:


- “Overview”
- “Message Source”
- “Message Type”
- “Message Severity”
- “Message Filters”
- “Simulation Handling”
- “Issuing Messages”
- “Shorthand Macros”
- “Filtering Messages”
- “Redirecting Message to File”
- “Promotion and Demotion”
- “Message Catcher”
- “Message Callbacks”
- “Stop Simulation Depending Upon Error Number”

## Overview

Transactors, scoreboards, assertions, environment and testcases use messages to report any definite or potential errors detected. They might also issue messages to indicate the progress of the simulation or provide additional processing information to help diagnose problems.

To ensure a consistent look and feel to the messages issued from different sources, you should use a common message service. It only concerns a message service with the formatting and issuance of messages, not their cause. For example, the time reported in a message is the time at which the message was issued, not the time a failed assertion started.

VMM message service uses the following concepts to describe and control messages:

- **Source**: component where the message is issued.
- **Type**: used to determine the message verbosity. For instance a message can be a note, a trace or a debug trace. Depending upon this verbosity, this message can be filtered out.
- **Severity**: used to determine the message severity. For instance a message can be a warning, an error or a fatal.
- **Handling**: used to determine the action associated to a given message. For instance stop the simulation after a fatal, count the number of errors, etc.
- **Filters**: used to promote or demote a message. For instance an error can be demoted to a warning or promoted to a fatal.

## Message Source

Each instance of the message service interface object represents a message source. A message source can be any component of a testbench: a command-layer transactor, a sub-layer of the self-checking structure, a testcase, a generator, a verification IP block or a complete verification environment. Messages from each source can be controlled independently of the messages from other sources.

## Message Type

Individual messages are categorized into different types by the author of the code used to issue the message. Assigning messages to their proper type lets a testcase or simulation produce and save only (or all) messages that are relevant to the concerns addressed by a simulation. Table 7-1 summarizes the available message types and their intended purposes:

*Table 7-1 Message Types*

| Message Type | Purpose |
|---|---|
| vmm_log::FAILURE_TYP | An error has been detected. The severity of the error is categorized by the message severity. |
| vmm_log::NOTE_TYP | Normal message used to indicate the simulation progress. |
| vmm_log::DEBUG_TYP | Message used to provide additional information designed to help diagnose the cause of a problem. Debug messages of increasing detail are assigned lower message severities. |
| vmm_log::TIMING_TYP | A timing error has been detected (for example, set-up or hold violation). |
| vmm_log::XHANDLING_TYP | An unknown or high-impedance state has been detected or driven on a physical signal. |

| Message Type | Purpose |
|---|---|
| vmm_log::REPORT_TYP<br>vmm_log::PROTOCOL_TYP<br>vmm_log::TRANSACTION_TYP<br>vmm_log::COMMAND_TYP<br>vmm_log::CYCLE_TYP | Additional message types that can be used by transactors. |
| vmm_log::INTERNAL_TYP | Messages from the VMM base classes should not be used when implementing user-defined extensions. |

## Message Severity

Individual messages are categorized into different severities by the author of the code used to issue the message. A message's severity indicates its importance and seriousness and must be chosen with care. For fail-safe reasons, certain message severities cannot be demoted to arbitrary severities. Table 7-2 summarizes the available message severities and their meaning:

*Table 7-2 Message Severities*

| Message Severity | Indication |
|---|---|
| vmm_log::FATAL_SEV | The correctness or integrity of the simulation has been compromised. By default, simulation is aborted after a fatal message is issued. Fatal messages can only be demoted into error messages. |
| vmm_log::ERROR_SEV | The correctness or integrity of the simulation has been compromised, but simulation may be able to proceed with useful result. By default, error messages from all sources are counted and simulation aborts after a certain number are observed. Error messages can only be demoted into warning messages. |
| vmm_log::WARNING_SEV | The correctness or integrity of the simulation has been potentially compromised, and simulation can likely proceed and still produce useful result. |

| Message Severity | Indication |
|---|---|
| vmm_log::NORMAL_SEV | This message is produced through the normal course of the simulation. It does not indicate that a problem has been identified. |
| vmm_log::TRACE_SEV | This message identifies high-level internal information that is not normally issued. |
| vmm_log::DEBUG_SEV | This message identifies medium-level internal information that is not normally issued. |
| vmm_log::VERBOSE_SEV | This message identifies low-level internal information that is not normally issued. |

## Message Filters

Filters can prevent or allow a message from being issued. Filters are associated and disassociated with message sources. They are applied in order of association and control messages based on their identifier, type, severity or content. Message filters can promote or demote messages severities, modify message types and their simulation handling. After a message has been subjected to all the filters associated with its source, its effective type and severity may be different from the actual type and severity originally specified in the code used to issue a message.

## Simulation Handling

Different messages require different action by the simulator once the message has been issued. Table 7-3 summarizes the available message handling and their default trigger:

*Table 7-3 Simulation Handlings*

| Simulation Handling | Action |
|---|---|
| vmm_log::ABORT_SIM | Terminates the simulation immediately and returns to the command prompt, returning an error status. This is the default handling after issuing a message with a *vmm_log::FATAL_SEV* severity. |
| vmm_log::COUNT_ERROR | Counts the message as an error. If the maximum number of such messages from all sources has exhausted a user-specified threshold, the simulation is aborted. This is the default handling after issuing a message with an *vmm_log::ERROR_SEV* severity. |
| vmm_log::STOP_PROMPT | Stops the simulation immediately and return to the simulation runtime-control command prompt. |
| vmm_log::DEBUGGER | Stops the simulation immediately and start the graphical debugging environment. |
| vmm_log::DUMP_STACK | Dumps the callstack or any other context status information and continue the simulation. |
| vmm_log::CONTINUE | Continues the simulation normally. |

## Shorthand Macros

A simple way of issuing messages can be achieved with macros. These macros provide a shorthand notation for issuing single-line failure messages.

Available shorthand macros are:

- ‘vmm_normal(log, str)
- ‘vmm_trace(log, str)
- ‘vmm_debug(log, str)
- ‘vmm_warning(log, str)

- ‘vmm_error(log, str)
- ‘vmm_fatal(log, str)

*Example 7-9 Using a Macro to Issue a Message*

```
‘vmm_error(this.log, "Unable to write to TxBD.TxPNT");
```

VCS provides the *$psprintf()* function that returns the formatted string instead of writing it into a string, like *$sformat()* does. You can use this function with the message macros, to display messages with runtime formatted content. The macros are designed to invoke the *$psprintf()* function only if you will issue the message as per this recommendation.

*Example 7-10 Using a Macro and the $psprintf() System Function*

```
`vmm_debug(this.log,
           $psprintf("Transmitting frame...%s",
                     fr.psdisplay("   ")));
```

## Issuing Messages

This section describes how to issue messages from within transactors, data and transaction models, the self-checking structure, the verification environment itself or testcases.

Issuing messages is simply done by instantiating a `vmm_log` object and using its methods `log::start_msg(), log::text(), log::end_msg()`.

Do not use *$display()* to manually produce output messages. If you must invoke a predefined method that produces output text (such as, the *vmm_data::psdisplay()* method), do so within the context of a message.

Example 7-11 shows how to issue a message with DEBUG severity. It is similar to Example 7-10.

*Example 7-11 Issuing a Message with Externally-Displayed Text*

```
vmm_log log = new(...);
...
if (log.start_msg(vmm_log::DEBUG_TYP,
                  vmm_log::TRACE_SEV)) begin
   log.text("Transmitting frame...");
   log.text(fr.psdisplay("   "));
   log.end_msg();
end
```

## Filtering Messages

It is possible to filter out messages based on their specific type and severity. The default severity of vmm_log can be set globally using a run time switch

```
+vmm_log_default=<sev>
```

where "<sev>" is the desired minimum severity and is a one of the following: "error", "warning," "normal," "trace," "debug" or "verbose".

Note: This switch affects all the vmm_log instances present in the verification environment.

There are two methods for filtering out specific vmm_log instances:

- Disable specified type of vmm_log messages from specified vmm_log instance using the vmm_log::disable_types() method.
- Set the minimum verbosity to the specified vmm_log instance, so that the severities above the specified levels are disabled. This is achieved by using the set_verbosity() method

For example, if the verbosity is set to NORMAL, the remaining TRACE, DEBUG, VERBOSE & DEFAULT severities are disabled.

Example 7-12 shows a simple use model:

*Example 7-12 Filtering Out Message by Type or Verbosity*

```
program automatic P;
class A;
  vmm_log log = new("SEQ_GT_COLLECTOR", "seq_cltr");
  task call_msg();
  begin
    `vmm_warning(log, "Warning: Hello collected");
    `vmm_error(log, "Error: Hello collected");
  end
  endtask
endclass

vmm_log log=new("Top", "program");
A a;
initial begin
  a = new;

  // Disable message type of all "SEQ_GT_COLLECTOR"
  // instances to a FAILURE
  log.disable_types(vmm_log::FAILURE_TYP,
                    "SEQ_GT_COLLECTOR", "seq_cltr",);

  // Change message verbosity of all "SEQ_GT_COLLECTOR"
  // instances to an ERROR
  log.set_verbosity(vmm_log::ERROR_SEV,
                   "SEQ_GT_COLLECTOR", "seq_cltr",);

  a.call_msg();
end
endprogram
```

Note: You can globally force the minimum severity level with `+vmm_force_verbosity=<sev>` runtime command-line option.

## Redirecting Message to File

You can issue messages to a separate file instead of sending them to a simulation log file. This way it is easier to trace/debug `vmm_log` messages. You can stop and start this logging at any point of simulation. There are two `vmm_log` base class methods called `log_start`() and `log_stop`() used to meet such requirements.

As a first step, you must disable this standard output. Use `log_stop()` method:

```
log.log_stop(vmm_log::STDOUT);
```

Then, the file handle can be passed to `log_start()` method:

```
log.log_start(file_handle);
```

*Example 7-13 Redirecting Message to File*

```
program automatic test ;
  vmm_log log = new("program","Test");

  initial begin
    int log_descr = $fopen("my_vmm.log");
    //Redirect messages to file my_vmm.log
    log.log_stop(vmm_log::STDOUT, "program","Test");
    if (log_descr == 0)
      `vmm_error(log,"Failed to $fopen ");
   else
     log.log_start(log_descr, "program","Test");

    `vmm_error(log, "message redirected to a file") ;

    //Redirect messages to STDOUT
```

```
    log.log_stop(log_descr, "program","Test");
    `vmm_error(log, "message in STDOUT") ;
  end
endprogram
```

## Promotion and Demotion

You can promote or demote messages. This means that you can promote a *warning* to an *error* or demote it to a *note*. This feature is useful for getting rid of expected failures like error injection or for changing the severity level of a given transactor.

A typical situation is to stop simulation on specific severities (such as, `ERROR_SEV`, which is the severity used by `` `vmm_error `` macro). This can be of interest for debugging a given error.

To configure all `vmm_log` objects to stop on error, use the `vmm_log::modify()` method:

```
log.modify("/./", "/./", 0,
           vmm_log::ALL_TYPS ,
           vmm_log::FATAL_SEV + vmm_log::ERROR_SEV,
           "/./",
           vmm_log::UNCHANGED,
           vmm_log::UNCHANGED,
           vmm_log::DEBUGGER);
```

Note: The last argument specifies the `log` to call the debugger.

For details, see `vmm_log::modify()` in Annex A.

## Message Catcher

In some cases, you might want your environment to execute specific code whenever a given message is issued by any of its components.

VMM provides an easy and flexible mechanism to do that using the `vmm_log_catcher` class. This class is based on `regexp` to specify matching `vmm_log` messages. When a message that matches a specified `regexp` is issued during simulation, the code that you specify gets executed.

`vmm_log_catcher` class comes with the following methods:

```
vmm_log_catcher::caught()
vmm_log_catcher::throw()
vmm_log_catcher::issue()
```

`vmm_log_catcher::caught()` method can be used to modify the caught message, change its type and severity. You can choose to ignore this message in which case it will not be displayed. The message can be displayed as is after executing your specified code. The updated message can be displayed by calling `vmm_log_catcher::issue` in the caught method.

The caught message, modified or unmodified, can be passed to other catchers that have been registered using the `vmm_log_catcher::throw` function. The messages to be caught are registered with the `vmm_log` class using the `vmm_log::catch` method.

To catch messages, first you need to extend the `vmm_log_catcher` class and implement its `caught(), throw()` and `issue()` method.

*Example 7-14 Extending and Customizing VMM Log Catcher*

```
class error_catcher extends vmm_log_catcher;
  virtual function void caught(vmm_log_msg msg);
    msg.text[0] = {" Acceptable Error" , msg.text[0]};
    msg.effective_severity = vmm_log::WARNING_SEV;
    issue(msg);
  endfunction
```

```
endclass
```

Next step is to instantiate your custom log catcher in your environment.

*Example 7-15 Registering Custom VMM Log Catcher for Specific Instance*

```
initial begin
  env = new();
  error_catcher catcher = new();
  env.build();
  catcher_id = env.sb.log.catch(catcher,,,1,,
                                vmm_log::ERROR_SEV,
                                "/Mismatch/");
  env.run();
end
```

In the previous example, the `error_catcher` class extends the `vmm_log_catcher` class and implements the caught method. The caught method prepends "Acceptable Error" to the original message and changes the severity to `WARNING_SEV`. In the initial block of the program block, an object of `error_catcher` is created and a handle is passed to catch method to register the catcher.

Any `vmm_log` message from scoreboard (sb), having `ERROR_SEV` as severity and including the string "Mismatch", will be caught and changed to `WARNING_SEV` with "Acceptable Error" prepended to it.

If you need the message to catch from all `vmm_log` instances, you can call catch with more arguments so that the pattern matching applies to all instances.

*Example 7-16 Registering Custom VMM Log Catcher for All Instances*

```
initial begin
  env = new();
  error_catcher catcher = new();
  env.build();
  catcher_id = env.sb.log.catch(catcher,/./,/./,1,,
                                vmm_log::ERROR_SEV,
```

"/Mismatch/");

```
 env.run();
end
```

To unregister a catcher, you can use, `vmm_log::uncatch(catcher_id)` or `vmm_log::uncatch_all()` methods. After a log catcher is unregistered using `uncatch` or `uncatch_all`, subsequent messages will not be caught and the user-defined extensions will no longer apply.

## Message Callbacks

The Message Service Class provides an efficient way of controlling the simulation and debugging your environment when certain messages are issued.

`vmm_log` provides pre-defined callbacks that are defined in `vmm_log_callbacks` object. Callbacks are associated with the message service itself, not a particular message service instance.

The available virtual methods are:

```
vmm_log_callbacks::pre_abort()
vmm_log_callbacks::pre_stop()
vmm_log_callbacks::pre_debug()
```

The `vmm_log_callbacks::pre_abort()` callback method is invoked by message service before simulation is aborted because of,

- `ABORT` simulation handling of particular vmm_log instance

- Exceed maximum number of COUNT_ERROR messages and on the basis of that you want to do some debug action /(print some logistics report).

Example 7-17 shows how to extend vmm_log_callbacks so that a specific action is taken when the `vmm_fatal is fired off.

*Example 7-17 Using vmm_log_callbacks::pre_abort Callback*

```
`include "vmm.sv"
program automatic test_log;

class cb extends vmm_log_callbacks;
  virtual function void pre_abort(vmm_log log);
    `vmm_note(log, "pre_abort cb has been invoked");
  endfunction
endclass

initial begin
  vmm_log log = new("", "");
  cb cb0 = new;
  log.append_callback(cb0);

  `vmm_fatal(log, "Aborting...");
end // initial begin
endprogram
```

Message service invokes vmm_log_callbacks::pre_stop() callback method before simulation is stopped due to STOP simulation handling of particular vmm_log instance.

Message service invokes vmm_log_callbacks::pre_debug() callback method before simulation is stopped due to DEBUGGER simulation handling of particular vmm_log instance.

### Stop Simulation Depending Upon Error Number

The Message Service Class provides an efficient way of stopping the simulation after a defined number of errors.

This is made possible with the `vmm_log.stop_after_n_errors()` method that allows to change the error threshold (10 by default).

*Example 7-18 Changing vmm_log Error Threshold*

```
program automatic test;
  vmm_log log = new("Test", "Errors");
  initial begin
    log.stop_after_n_errors(50) ;
    for (i=1 ; i<100 ; i=i+1)
    `vmm_error(log, $psprintf ("*** Error No. %0d ***\n"
, i ));
  end
endprogram
```

## Class Factory Service

This section contains the following topics:

- “Overview”
- “Modeling a Transaction to be Factory Enabled”
- “Creating Factories”
- “Replacing Factories”
- “Factory for Parameterized Classes”
- “Factory for Atomic Generators”
- “Factory for Scenario Generators”

- “Modifying a Testbench Structure Using a Factory”

## Overview

Factory Service provides an easy way to replace any kind of object, transaction, scenario, or transactor by a similar object. This replacement can take place from anywhere in the verification environment or in the test case.

The following typical situations are for object oriented extensions:

- Replace a class by a derived class.
- Replace a parameterized class by a derived class.
- Replace a transaction modeled using `vmm_data` by a derived class.
- Replace a scenario extending `vmm_scenario` by another scenario.
- Replace a transactor modeled using `vmm_xactor` or `vmm_group` by a derived transactor.

Similarly, it is possible to use factory to replace similar objects between classes:

- Switch configurations in transactors.
- Switch scenarios in generators.

Factory service acts as a replacement for object construction. Rather than declaring an object and constructing it using its `new()` method, VMM provides facilities to consider objects as factory.

The factory service use model is as follows:

- Implement any object as a class and use `vmm_class_factory macro for having this object becoming factory enabled.
- Create object instance by using a method class::create_instance() instead of new(), this object instance in turn becomes a factory, for example, an object that can be replaced.
- Replace this factory from wherever the replacement is desired, such as a parent transactor, environment or a testcase. This is achieved by using a set of static methods for either copying or allocating a new object using, class::override_with_copy() or class::override_with_new().

Factory service is very handy for modeling generators. Usually generators declare a transaction and then randomize it by applying its built-in constraints.

When other set of constraints should be applied to this transaction, you can replace this transaction by a new transaction that derives the latter one.

You can easily carry out these steps by declaring the generator transaction with class::create_instance() and replacing it in your test with class::override_with_new() method.

Certainly, you should be careful regarding phases where factory should be created and replaced. Creation should take place in `start_of_sim` phase and recording the replacement should take place in a preliminary phase like `configure_test`.

*Table 7-4 Phases for Factory Creation and Replacement*

| Factory creation and replacement | Phases for {Transactions, Transactors, objects, MSS} | Phases for generators such as {Atomic, single stream Scenario, MSS} |
|---|---|---|
| `create_instance()` | `start_of_sim_ph()` | `N/A (built-in)` |
| `override_with_()` | `configure_test_ph()` | `configure_test_ph()` |

The following sections provide more details on how to model, add and replace factories.

## Modeling a Transaction to be Factory Enabled

This section explains how to model a transaction so that it can be considered as a factory, either in transactor or in the verification environment.

This requires that the class implements a general-purpose constructor, `allocate()` and `copy()` methods.

Note: These methods are automatically implemented with `vmm_data` extension while using shorthand macros.

As any class factory is mostly based on user-friendly macros, replacing it by an extended class requires the following guidelines:

- Provide a general-purpose constructor. The constructor must have the default arguments, so that the calls to `new()` are allowed. If some specific members need to be initialized to user specific values, `set*`/`get*` methods can be used to handle these assignments. Another approach is to use advanced options as described in following section.
- Create a new object by using the `allocate()` method. In this case, the extended class provides the necessary implementation to allocate data members, subsequent objects, etc.
- Create a new object by using the `copy()` method. In this case, the extended class provides the necessary implementation to copy data members, subsequent objects, etc.

Example 7-19 shows how to model a simple transaction that extends `vmm_data`.

*Example 7-19 Factory Enabled Transaction*

```
class cpu_trans extends vmm_data;
   `vmm_typename(cpu_trans);
   typedef enum bit {READ = 1'b1, WRITE =  1'b0} kind_e;
   rand bit [31:0] address;
   rand bit [7:0]  data;
   rand kind_e    kind;
   rand bit [3:0] trans_delay;

   `vmm_data_member_begin(cpu_trans)
   `vmm_data_member_scalar(address, DO_ALL)
   `vmm_data_member_scalar(data, DO_ALL)
   `vmm_data_member_scalar(trans_delay, DO_ALL)
   `vmm_data_member_enum(kind, DO_ALL)
   `vmm_data_member_end(cpu_trans)

    `vmm_class_factory(cpu_trans)
endclass
`vmm_channel(cpu_trans)
```

Note: `vmm_typename() creates the get_typename() function that contains a typename to return a string like “cpu_trans”. This is very convenient for displaying this object type.

You should use shorthand macros to model data members and `vmm_class_factory declares all necessary methods required to turn this transaction into a factory. Shorthand macros vmm_data_member_* automatically implement allocate() and copy() methods.

In case you need to add extra content to this transaction such as, new members, constraints, and methods, you just extend its base class.

Example 7-20 shows how to model a simple transaction that extends cpu_trans. The only required step is to add a `vmm_class_factory statement at the end of this transaction to make the class factory ready.

*Example 7-20 Factory Enabled Derived Transaction*

```
class test_write_back2back_test_trans extends cpu_trans;
  `vmm_typename(test_write_back2back_test_trans)
   // Macros which define utility methods like
   // copy, allocate, etc
  `vmm_data_member_begin(test_write_back2back_test_trans)
  `vmm_data_member_end(test_write_back2back_test_trans)
  constraint cst_dly {
     kind == WRITE;
     trans_delay == 0;
  }
  `vmm_class_factory(test_write_back2back_test_trans)
endclass
```

## Creating Factories

The previous section explains how to model a transaction so that it can be considered as a factory. This section describes how to instantiate this object in a transactor.

Usually, an object is declared in a transactor and constructed in its `new()` task. This modeling style does not apply for factories. A factory must be explicitly created in `start_of_sim` phase.

Note: The `create_instance()` method is static and must be prefixed with its class name.

Example 7-21 shows how to instantiate and use the previously created transaction factory in a Multi-Stream Scenario (MSS). The scenario has to be instantiated in `start_of_sim` phase.

*Example 7-21 Instantiation of Transaction Factory in MSS*

```
class cpu_rand_scenario extends vmm_ms_scenario;
  cpu_trans       blueprint;
  `vmm_scenario_new(cpu_rand_scenario)
  `vmm_scenario_member_begin(cpu_rand_scenario)
  `vmm_scenario_member_vmm_data(blueprint, DO_ALL,
                                DO_REFCOPY)
  `vmm_scenario_member_end(cpu_rand_scenario)
  function new();
     blueprint = cpu_trans::create_instance(this,
                                            "blueprint",
                                      `__FILE__, `__LINE__);
  endfunction

  virtual task execute(ref int n);
     cpu_trans tr;
     bit res;
     vmm_channel chan;
     if (chan == null) chan = get_channel("cpu_chan");
     $cast(tr, blueprint.copy());
     res = tr.randomize();
```

```
      chan.put(tr);
    endtask

    `vmm_class_factory(cpu_rand_scenario)
endclass
```

## Replacing Factories

The factory is now available in the transactor, so you can use it as is or replace in the test, either by copying it from another transaction or by constructing it from scratch.

Both use models are made possible using the `class_name::override_with_copy()` or `class_name::override_with_new()` functions.

Example 7-22 shows how to add two transactors to a program block and use the default factory, i.e. `cpu_trans`.

*Example 7-22 Instantiation of Transactor Factory*

```
class env extends vmm_group;
   ahb_gen gen0, gen1;

   virtual task build_ph();

      gen0 = ahb_gen::create_instance(this, "gen0");
      gen1 = ahb_gen::create_instance(this, "gen1");

      vmm_log log = new("prgm", "prgm");
      `vmm_note(log, {"gen0.tr.addr=", gen0.tr.addr);
   endtask
endclass
```

To replace a factory by another instance of the same type with different data member values, you can use the `class_name::override_with_copy()` method with a regular expression that matches a specific pattern, either in the `vmm_object` hierarchy or by referring to the transactor structure.

Example 7-23 shows how to replace a specific test that replaces initial factory with a copy.

*Example 7-23 Replacement of Transaction Factory*

```
class test_read_back2back extends vmm_test;

  function new(string name);
     super.new(name);
  endfunction

  virtual function void configure_test_ph();
     test_read_back2back_test_trans tr = new();
     tr.address = 'habcd_1234;
     tr.address.rand_mode(0);
     cpu_trans::override_with_copy("@%*", tr, log,
                                   `__FILE__, `__LINE__);
  endfunction
endclass
```

To replace a factory by a derived class, you can use the `class_name::override_with_new()` method with a regular expression that matches a specific pattern, either in the `vmm_object` hierarchy or by referring to the transactor structure.

In the case of referring to the transactor structure, the new transaction type for factory replacement should be considered and returned by `test_read_back2back_test_trans::this_type`.

This transaction type is usually a derived class, since the `class_name::create_instance()` method considers its underlying base class by default, so there is no point in using a statement such as,

```
cpu_trans::override_with_new("@%*",
   test_write_back2back_test_trans::this_type(),
   log, `__FILE__, `__LINE__);
```

Example 7-24 shows how to replace the initial factory with a derived object.

*Example 7-24 Replacing Derived MSS in Test*

```
class test_write_back2back extends vmm_test;
  function new(string name);
     super.new(name);
  endfunction
  virtual function void configure_test_ph();
     cpu_trans::override_with_new("@%*",
       test_write_back2back_test_trans::this_type(), log,
        `__FILE__, `__LINE__);
  endfunction
endclass
```

Note: The factory replacement takes place in the `test2::start_of_sim` phase. This is necessary as this should always be called before `ahb_gen::config_dut` phase, otherwise subsequent calls to `class_name::override_with_new()` are not considered.

## Factory for Parameterized Classes

Factories are general purpose and apply to any kind of object. Modeling transactions can be achieved either by extending `vmm_data`, `vmm_object` or no object at all.

Example 7-25 shows how to write a factory on top of a parameterized class.

*Example 7-25 Parameterized Class Factory*

```
program P;
class cpu_trans #(type T=int) extends vmm_data;
  `vmm_typename(cpu_trans#(T))
  T value;
  `vmm_data_member_begin(cpu_trans#(T))
  `vmm_data_member_end(cpu_trans#(T))
  `vmm_class_factory(cpu_trans#(T))
endclass

class cpu_gen #(type T=int) extends vmm_xactor;
   `vmm_typename(cpu_gen#(T))
   cpu_trans #(T) tr;
   function new(string inst, vmm_unit parent=null);
     super.new("cpu_driver", inst, , parent);
    tr = cpu_trans#(T)::create_instance(this, "MY_TRANS");
     tr.display();
   endfunction
endclass

class my_cpu_trans #(type T=int) extends cpu_trans#(T);
  `vmm_typename(my_cpu_trans#(T))
  `vmm_data_member_begin(my_cpu_trans#(T))
  `vmm_data_member_end(my_cpu_trans#(T))
  T value;
  `vmm_class_factory(my_cpu_trans#(T))
endclass

class tb_env extends vmm_group;
   cpu_gen#(string) gen;
   function new(string inst = "env");
     super.new("tb_env", inst);
   endfunction
   virtual function void build_ph();
      gen = new("gen", this);
   endfunction
endclass
```

```
class my_test extends vmm_test;
   my_cpu_trans#(string) mtr;
   function new(string inst);
     super.new(inst);
   endfunction
   function void configure_test_ph();
     cpu_trans#(string)::override_with_new("@%*",
                    my_cpu_trans#(string)::this_type, log);
   endfunction
endclass

tb_env env;
my_test tst;
initial begin
   env = new("env");
   tst = new("test");
   vmm_simulation::run_tests();
end

endprogram
```

## Factory for Atomic Generators

Atomic generators are used to randomize unrelated transactions and posting them to a `vmm_channel`. By default, `atomic_gen` comes with a transaction blueprint named `randomized_obj` that can be replaced by a factory. This factory can have the same type as `randomized_obj` or be an extension of it.

Consider an example where `atomic_gen` is wrapped in a `vmm_xactor`. This is necessary to ensure its run flow is properly controlled.

*Example 7-26 Creating an Atomic Generator Using Factory*

```
class ahb_env extends vmm_xactor;
   `vmm_typename(env)
   cpu_trans_atomic_gen gen;
```

```
   function new(string name);
      super.new(get_typename(), name);
   endfunction

   virtual function void build_ph();
      gen = ahb_gen::create_instance(this,"gen",);
   endfunction

endclass
```

In the test, it is possible to directly replace `atomic_gen::randomized_obj` by a factory using `override_with_new` for a given generator.

Similarly, a copy of `atomic_gen::randomized_obj` can be overridden in the other generator by also passing its implicit name to the `override_with_copy()` method.

Example 7-27 shows how to replace an `atomic_gen::randomized_obj` factory in specific generator by its name.

*Example 7-27 Overriding Atomic Scenario Factory*

```
class test extends vmm_test;
   cpu_trans mtr;
   virtual function void start_of_sim_ph();
      // Replace factory in env0.gen
      cpu_trans::override_with_new(
         "@env0:gen:randomized_obj",
         my_cpu_trans::this_type, log, `__FILE__,
         `__LINE__);

      // copy factory in env1.gen
      mtr = new; mtr.addr = 'h55;
      cpu_trans::override_with_copy(
         "@env1:gen:randomized_obj", mtr, log, `__FILE__,
         `__LINE__);
```

```
   endfunction
endclass: test
```

## Factory for Scenario Generators

Scenario generators are aimed at randomizing lists of related transactions and posting them to a `vmm_channel`. By default, `scenario_gen` comes with a scenario blueprint that you can replace by a factory. You can make this factory of the same type as the scenario or an extension of it.

As described in preceding sections, scenarios are similar to any kind of transaction and need to implement general-purpose `new()`, `allocate()` and `copy()` methods. These methods are directly invoked by the `override_with_copy()` and `override_with_new()` methods. The only required step is to add a `vmm_class_factory` macro at the end of the scenario to make this class factory ready.

Example 7-28 shows how to model a scenario.

*Example 7-28 Modeling Scenario Factory*

```
class test_scenario extends my_scenario;
   int TST_KIND;
   constraint cst_test {
      scenario_kind == TST_KIND;
      foreach (items[i]) {
         items[i].data == 'ha5a5a5a5;
      }
   }
   function new();
      TST_KIND = define_scenario("tst_scenario", 3);
   endfunction

   function vmm_data allocate();
      test_scenario scn = new;
```

```
      allocate = scn;
   endfunction

   function vmm_data copy(vmm_data to = null);
      test_scenario scn = new;
      scn.TST_KIND = this.TST_KIND;
      scn.stream_id = this.stream_id;
      scn.scenario_id = this.scenario_id;
      copy = scn;
   endfunction
   `vmm_class_factory(test_scenario)
endclass
```

Similarly to other transactors, `scenario_gen` should be wrapped in a `vmm_xactor`. This is necessary to ensure its run flow is properly controlled and that you properly create the factory using a two-phase approach.

Example 7-29 shows how to wrap a `scenario_gen` into a controllable `vmm_xactor`, where `gen.my_scenario` is the factory.

*Example 7-29 Overriding Scenario Factory*

```
class env extends vmm_group;
   `vmm_typename(env)
   cpu_trans_scenario_gen gen;
   my_scenario scn;

   function new(string name);
      super.new(get_typename(), name);
   endfunction

   virtual function void build_ph();
      gen = new(this,"gen");
      scn = cpu_trans_scenario::create_instance(
                                       this,"scn");
   endfunction

   virtual function void connect_ph();
      gen.register_scenario("my_scenario", scn);
```

```
   endfunction
endclass
```

In the test, it is now possible to directly replace `gen:my_scenario` by a factory using `override_with_new`. This is made possible by simply passing the generator's hierarchical name to this method.

Similarly, a copy of `my_scenario` can be overridden in the scenario generator by passing its implicit name to the `override_with_copy()` method.

Example 7-30 shows how to replace `vmm_scenario_gen` factory by its name.

*Example 7-30 Overriding Scenario Generator Factory*

```
class test extends vmm_test;
   my_scenario other_scn;

   virtual function void start_of_sim_ph();
   // replace factory in env0.gen with new scenario
      my_scenario::override_with_new(
         "@env0:gen:my_scenario",
         test_scenario::this_type,
         log, `__FILE__, __LINE__);

      // copy factory in env1.gen
      other_scn = new;
      my_scenario::override_with_copy(
         "@env0:gen:my_scenario", other_scn, log,
         `__FILE__, `__LINE__);
   endfunction
endclass: test
```

## Modifying a Testbench Structure Using a Factory

Because the test timeline executes after the pre-test timeline, a test cannot use the `override_with_new()` or `override_with_copy()` factory methods to modify the structure of an environment.

By the time the test timeline starts to execute, the environment will already have been built during its `"build"` phase and all of the testbench component instances will already have been created, so subsequent calls to `override_with_new()` or `override_with_copy()`.So you do not consider them.

A test can only use the factory replacement methods to affect the instances generators dynamically create. A test must use `vmm_xactor_callbacks` to affect the behavior of testbench components, not factories.

Implementation does not cause this limitation. However, it arises from requirements for test concatenation. When concatenating multiple tests, a test must be able to restore the environment to its original state. It is simple to do so by removing callback extensions, but it is not possible to do so if you construct the environment with a test-specific instance.

However, to simplify the use model when not using test concatenation, you execute the `vmm_test::set_config()` method  before the phasing of the pre-test timeline.

It is thus possible for a test to set factory instances by using the `override_with_new()` or `override_with_copy()` factory methods. However, it is not possible to concatenate such a test with

other tests, as its modification of the environment would interfere with the configuration of other tests. You invoke this method only if there is only one test selected for execution.

# Options & Configurations Service

This section contains the following topics:


- “Overview”
- “Hierarchical Options (vmm_opts)”
- “Structural Configurations”
- “RTL Configuration”


## Overview

VMM comes with comprehensive ways of configuring transactors, components and verification environments.

You can use,

- Hierarchical options to get options from command line, command file or in the VMM code directly.
- Structural options to configure transactors and ensure their configuration are well set in the configure phase in a given phase called configure.
- RTL configuration to configure both RTL and verification environment.

## Hierarchical Options (vmm_opts)

Configurations can be set from the simulator command line or a file. You can set them on an instance basis or hierarchically by using regular expressions.

Configuration parameters can be set from three different sources, in order of increasing priority: within the code itself using `vmm_opts::set_*()` methods, external option files and command-line options.

You can either generate configuration parameters through randomization or set with hierarchical/global options by calling `vmm_opts::get_*()` methods.

The following methods let you specify configurations for,

- Global configurations with the following expressions to set ***Field=Value*** for all objects that contain option ***Field***. Note that simv is the name of the executable,

  ```
  simv +vmm_opts+Field=Value
  ```

- Hierarchical objects by using the following expressions to set ***Field=Value*** for unique object ***Top0.instance,***

  ```
  simv +vmm_opts+Field=Value@Top0.instance
  ```

- Hierarchical objects by using the following expressions to set ***Field=Value*** for all objects under ***Top0*** that contain option ***Field,***

  ```
  simv +vmm_opts+Field=Value@Top0
  ```

## Specifying Placeholders for Hierarchical Options

Configurations are usually modeled as a class and correspond to a container where all possible options are defined as data members.

The static methods `vmm_opts::get_object_*` assign a specific data member with a value from the `vmm_opts::set_*` methods.

Example 7-31 shows a configuration of two members: boolean ***b*** and integer ***i*** are flagged with ***B*** and ***I*** tags respectively, and the ***is_set*** variable is set when the option is overridden from command line. This is handy to find out whether a used value is a default one or comes from the command line.

*Example 7-31 Adding Options in a Class*

```
class vip extends vmm_xactor;
   bit b;
   int i;
   function configure_ph();
      bit is_set;
      b = vmm_opts::get_object_bit(is_set, this, "B",
                                   "SET b value", 0);
     i = vmm_opts::get_object_int(is_set, this, "I", 0,
                                   "SET i value", 0);
   endfunction
endclass
```

## Setting Hierarchical Options

Configurations can be set from the simulator command line or a file. You can set them on an instance basis or hierarchically by using regular expressions.

Example 7-32 shows how to assign configuration members: boolean ***b*** and integer ***i*** in a test. This is made possible by using `vmm_opts::set_int` and `vmm_opts::set_bit` in the program block. Of course, these assignments could be anywhere in `vmm_timelines` or in `vmm_test:configure_test_ph()`.

*Example 7-32 Assigning Options in Code Block*

```
function build_ph();
   vip vip0 = new("vip0", null);
   vip vip1 = new("vip1", null);
endfunction

function start_ph();
   vmm_opts::set_bit("vip0:b",null);
   vmm_opts::set_int("vip0:i",null);
   vmm_opts::set_bit("vip1:b",null);
   vmm_opts::set_int("vip1:i",null);

   $display(" Value of vip0:b is %0b", vip0.b);
   $display(" Value of vip0:i is %0d", vip0.i);
   $display(" Value of vip1:b is %0b", vip1.b);
   $display(" Value of vip1:i is %0d", vip1.i);
endfunction
```

Note: It is also possible to set configurations that only belong to a given hierarchy, for instance the following line assigns ***B*** configurations for all ***b**** matching objects that are under the ***d2*** root object.

*Example 7-33 Setting Options Using Regular Expressions*

```
vmm_opts::set_int("%b*:B", 99, d2);
```

## Setting Hierarchical Options on Command Line

After you have defined the configurations, it is possible to change their values from,

- The simulator command line with ***+vmm_opts+Field=Value*** or ***+vmm_Field=Value***
- An option file with the following syntax for assigning ***d2:b1.b=88***, all ***d1.*.b=99***, ***i=1'b0*** globally and ***c2.b1.str="NEW_VAL2"***

*Example 7-34 Option File*

```
+B =88@d2:b1
+B =99@d1*
+I = 0
+STR=NEW_VAL2@c2:b1
```

The following example shows how its default values are returned when no options are specified on the command line:

```
% ./simv
Chronologic VCS simulator copyright 1991-2008
Contains Synopsys proprietary information.

 Value of vip0:b is 0
 Value of vip0:i is 0
 Value of vip1:b is 0
 Value of vip1:i is 0
Simulation PASSED on /./ (/./) at                    0
(0 warnings, 0 demoted errors & 0 demoted warnings)
           V C S   S i m u l a t i o n   R e p o r t
Time: 0
CPU Time:     0.020 seconds;      Data structure size:
0.0Mb
```

The following example shows how to globally assign values for boolean ***b=1'b1*** and integer ***i=10***:

```
% ./simv +vmm_opts+I=10 +vmm_B=1
Chronologic VCS simulator copyright 1991-2008
Contains Synopsys proprietary information.

 Value of vip0:b is 1
 Value of vip0:i is 10
 Value of vip1:b is 1
```

```
 Value of vip1:i is 10
Simulation PASSED on /./ (/./) at                    0
(0 warnings, 0 demoted errors & 0 demoted warnings)
           V C S   S i m u l a t i o n   R e p o r t
Time: 0
CPU Time:     0.030 seconds;      Data structure size:
0.0Mb
```

The following example shows how to assign values for boolean ***vip0.b=1'b1*** and integer ***vip1.i=10***:

```
% ./simv +vmm_opts+I=10@vip1 +vmm_B='1@*vip0'
Chronologic VCS simulator copyright 1991-2008
Contains Synopsys proprietary information.

 Value of vip0:b is 1
 Value of vip0:i is 0
 Value of vip1:b is 0
 Value of vip1:i is 10
Simulation PASSED on /./ (/./) at                    0
(0 warnings, 0 demoted errors & 0 demoted warnings)
           V C S   S i m u l a t i o n   R e p o r t
Time: 0
CPU Time:     0.020 seconds;      Data structure size:
0.0Mb
```

For details on all available options, see the VMM Reference Guide.

## Structural Configurations

Structural configuration is an important aspect of verification environment composition. This is usually required for dynamically building verification components based upon configurations specified either on the command line or in a command file. You can set these configurations on an instance basis or hierarchically by using regular expressions.

Configuration parameters that affect the structure of the environment itself you must set during the `"build"` phase and implement the `vmm_unit::build_ph()` method.

You can specify these configuration parameters using options, but you typically set using RTL configuration parameters. Because you invoke the `vmm_unit::build_ph()` methods in a top-down order, procedural parameter settings from higher-level modules supersede procedural parameter settings from lower-level modules.

Due to the nature of structural configurations, there is no need for automatic randomization of structural configuration parameters.

The use model of structural configuration is similar to hierarchical configurations except that specific `vmm_unit` shorthand macros must be used to instrument transactors that extend the `vmm_unit` base class.

You can set structural configuration parameters from three different sources, in order of increasing priority:

- within the code itself using `vmm_opts::set_*()` methods.
- external option files.
- command-line options.

You set these parameters by explicitly calling the `vmm_opts::get_*()` methods in `vmm_timeline` or environment.

The following use models are available for specifying a structural configuration:

- Global configurations by using the following expressions to set ***Field=Value*** for all objects that contain option ***Field,***

```
simv +vmm_opts+Field=Value
```

- Instance-specific objects by using the following expressions to set ***Field=Value*** for unique object ***Top0.instanceX,***

```
simv +vmm_opts+Field=Value@Top0.instanceX
```

- Hierarchical objects by using the following expressions to set ***Field=Value*** for all objects under ***Top0*** that contain option ***Field,***

```
simv +vmm_opts+Field=Value@Top0
```

## Specifying Structural Configuration Parameters in Transactors

Structural configuration declarations should sit in the transactor that extends `vmm_unit`.

You can use a pre-defined set of shorthand macros to attach structural configuration parameters to transactor structural tags, which you can access from either the command line or a command file. These macros automatically implement the declaration and assignment of structural options in the `build` phase.

The following ***vmm_unit*** shorthand macros are available:

```
`vmm_unit_config_int(    int_data_member,”doc",
                         def_value, transactor)
‘vmm_unit_config_boolean(boolean_data_member,"doc",
                         def_value, transactor)
‘vmm_unit_config_string( string_data_member,"doc",
                         def_value, transactor)
```

Example 7-35 shows a structural configuration where three data members: {boolean ***b***, integer ***i***, string ***s***} are tagged with the {***B***, ***I***, ***S***} keywords respectively.

*Example 7-35 Defining Structural Configurations*

```
class vip extends vmm_xactor;
   `vmm_typename(vip)
   int i;
   bit b;
   string s;

   function new(string inst, vmm_unit parent = null);
      super.new(get_typename(), inst, parent);
   endfunction

function void configure_ph();
   `vmm_unit_config_int(i,1,"doc",0,vip)
   `vmm_unit_config_boolean(b,"doc",0,vip)
   `vmm_unit_config_string(s,"doc", "null",vip)
endfunction
endclass
```

## Setting Structural Configuration Parameters

Structural configuration parameters can be set from the simulator command line or a file. You can set them on an instance basis or hierarchically by using regular expressions.

Example 7-36 shows how to assign the ***v1*** configuration members: {boolean ***b***, integer ***i***, string ***s***} in a test. This is made possible by using `vmm_opts::set_int` and `vmm_opts::set_bit`.

Certainly, these assignments could be anywhere in `vmm_timeline` or in `vmm_test:configure_test_ph()`. They have to be executed before the corresponding `vmm_opts::get_* methods/ vmm_unit_config` macro execution.

*Example 7-36 Setting Structural Configurations in Code Block*

```
function void configure_ph();
   vip v1;
   vmm_opts::set_bit("v1:b",1);
   vmm_opts::set_int("v1:i",2);
   vmm_opts::set_string("v1:s",”Burst”);

   v1 = new(this, "v1");

   $display("v1.i=%0d, v1.b=%0d”, v1.i, v1.b);
endfunction
```

## Setting Options on Command Line

After you have defined the configurations, it is possible to change their values from,

- The simulator command line with ***+vmm_opts+Field=Value*** or ***+vmm_Field=Value.***
- An option file with following syntax for assigning ***d2:b1.b=88***, all ***d1.*.b=99***, ***i=1’b0*** globally and ***c2.b1.str=”NEW_VAL2”***

```
+B =88@d2:b1
+B =99@d1*
+I = 0
+STR=NEW_VAL2@c2:b1
```

The following example shows how to assign values for boolean ***v1.b=1’b0*** and integer ***v1.i=9***:

```
./simv +vmm_b=0 +vmm_opts+i=9@v1
```

## RTL Configuration

RTL configuration is an important aspect for ensuring that the RTL and testbench share the same configuration. This can be handy for sharing parameters such as,

- Number of input ports for a given protocol.
- Number of output ports for a given protocol.
- Architectural parameters like FIFO sizes, DMA capabilities and IRQs.
- Latency, bandwidth limitations, etc.
- Specific operating modes.

As opposed to hierarchical or structural configurations, RTL configuration solely depends on an input file that describes available options for a given instance. This input file allows the testbench and RTL to share the same configuration.

The following key features are supported by this set of VMM base classes:

- Support configurable RTL.
- Support RTL configuration with randomized / directed parameters.
- Support functional coverage of configuration.
- Support composition of RTL configurations.
- Support multiple instances of the same RTL module with different configurations.

- Support partially-specified configurations.

RTL configuration is performed using compile-time conditional code (i.e. ***`ifdef/`endif***) or parameter values, all of which are set before simulation runs. It is therefore impossible to randomize RTL configuration in the same simulation run and also run the test that will verify that configuration.

You must use the following two-pass process:

- First pass to generate the RTL configuration to use. This can be manual or external to VCS.
- Second pass to verify that configuration. This pass might be repeated multiple times to apply multiple tests to the same configuration. During this pass RTL and testbench are compiled using RTL configuration as created in first pass.

The second pass must not depend on random stability to reproduce the same RTL configuration. Instead, it should depend on a configuration specification file that is read in to set the RTL configuration parameters. This enables the RTL configuration to be specified manually, not only randomly.

## Defining RTL Configuration Parameters

RTL configuration parameters should be declared in a transactor configuration that extends the `vmm_rtl_config` base class.

Note: This transactor configuration acts as a data member container and is not supposed to be run.

A pre-defined set of shorthand macros can be used to attach RTL configuration parameters to transactor RTL tags, which you can access from either the command line or a command file.

The following `vmm_rtl_config` shorthand macros are available:

```
`vmm_rtl_config_int    (RTL_config_name,
                        RTL_config_fname)
`vmm_rtl_config_boolean(RTL_config_name,
                        RTL_config_fname)
`vmm_rtl_config_string (RTL_config_name,
                        RTL_config_fname)
```

Example 7-37 shows how to model a configuration where RTL configuration parameters: {boolean ***mst_enable***, integer ***addr_width***} are tagged with {***mst_enable***, ***mst_width***} keywords respectively. By default, these data members can be randomized and associated with user-specific constraints.

*Example 7-37 Modeling RTL Configuration for Transactor*

```
class ahb_master_config extends vmm_rtl_config;
   rand  int addr_width;
   rand  bit mst_enable;
   string    kind = "MSTR";

   constraint cst_mst {
     addr_width == 64;
     mst_enable == 1;
   }
   `vmm_rtl_config_begin(ahb_master_config)
      `vmm_rtl_config_int(addr_width, mst_width)
      `vmm_rtl_config_boolean(mst_enable, mst_enable)
      `vmm_rtl_config_string(kind, kind)
   `vmm_rtl_config_end(ahb_master_config)

   function new(string name = "",
                vmm_rtl_config parent = null);
      super.new(name, parent);
   endfunction

endclass
```

## Using RTL Configuration in vmm_unit Extension

A transactor can simply refer to the RTL configuration by declaring a handle to this class that gets associated in the `vmm_unit::configure` phase.

You can use the static method `vmm_rtl_config::get_config` to handle this association.

Example 7-38 shows how to associate a previously declared ***ahb_master_config*** object within a transactor.

*Example 7-38 Retrieving a RTL Configuration in Transactor*

```
class ahb_master extends vmm_xactor;
   ahb_master_config cfg;

   function new(string name, vmm_unit parent = null);
      super.new(get_typename(), name, parent);
   endfunction

   function void configure_ph();
      $cast(cfg, vmm_rtl_config::get_config(this);
   endfunction
endclass
```

After you have instantiated the transactor in its enclosing environment, you must properly construct and associate it with the right RTL configuration file.

This assumes that a RTL configuration file with name like "***INST.rtlconfig***" was previously created using the ***+vmm_gen_rtl_config*** first pass (see the following section).

Example 7-39 shows how to map the previously declared ***ahb_master*** transactor with the right RTL configuration file name.

*Example 7-39 Mapping Transactor RTL Configuration in Environment*

```
class env extends vmm_group;
   ahb_master mstr;

   function new(string name, vmm_unit parent = null);
      super.new(get_typename(), name, parent);
   endfunction

   function void build_ph();
     mstr = new(this, "mst");
     env_cfg.map_to_name("^");
     env_cfg.mst_cfg.map_to_name("env:mst");
   endfunction
endclass
```

## First Pass: Generation of RTL Configuration Files

The first pass to generate the RTL configuration can take place after the transactor configuration is ready.

You activate the file generation when running the simulation with `+vmm_gen_rtl_config` option.

In this case, the simulator considers all objects that extend `vmm_rtl_config` base class. During this phase, all transactor configurations are created, randomized and their content is dumped to multiple RTL configuration files. No simulation is run during this pass.

The following example shows how to create RTL configuration files by prefixing all output files with '***RTLCFG'***:

```
% ./simv +vmm_rtl_config=RTLCFG +vmm_gen_rtl_config

% more RTLCFG:env_cfg:mst_cfg.rtl_conf
mst_width  : 64;
mst_enable : 1;
```

```
kind        : MSTR;
```

### Second Pass: Simulation Using RTL Configuration File

The following example shows how to kick off a simulation by reading all RTL configuration files that are prefixed with '***RTLCFG'***:

./simv +vmm_rtl_config=RTLCFG

## Simple Match Patterns

This section contains the following topics:

- "Overview"
- "Pattern Matching Rules"

### Overview

Simple match pattern performs hierarchical name matching in a specific hierarchical namespace. As `vmm_object` instance names are in the form of `top::subenv::vip`, writing usual regular expressions can be cumbersome and require to escape all delimiters consisting of `':'`. character.

To overcome this issue, VMM comes with a rich set of custom regular expressions. These expressions perform hierarchical name matching in a specific hierarchical namespace. Using this custom regular expression is turned on by simply appending the `'@'` character before the expression.

Here is a description of specific character that VMM regular expression interprets:

- "**:**" is used as hierarchical name separator, '**.**' character with no need to be escaped
- "**@**" is used to indicate a match pattern
- "**/**" is used for normal regular expressions

A match pattern matches every character as-is, except for meta-characters, which match in the following manner:

- "**.**" matches any one character, except '**:**'
- "*****" matches any number of characters, except '**:**'
- "**?**" matches zero or one character, except '**:**'
- "**%**" matches zero or more colon-separated names, including the final colon

## Pattern Matching Rules

*Table 7-5 VMM Regular Expression Pattern Matching Rules*

| Pattern | Description | Matches | Does Not Match |
|---|---|---|---|
| @%. | Matches any path ending with a single character as the last element | t<br>t:s<br>t:s:v | t:sub_env |
| @%* | Matches any hierarchical path | top<br>top:sub_env<br>top:sub_env:vip | top:<br>top:sub_env:<br>top:sub_env:vip: |

| | | | |
|---|---|---|---|
| @%? | Matches any hierarchical path, including null string | t<br>t:s<br>t:s:v<br>t ::v | top<br>top:sub_env<br>top:sub_env:vip |
| @top:*:vip | Matches the occurrence of any string | top:sub_env0:vip<br>top:sub_env1:vip | top:vip<br>top:sub_env0:slice0:vip |
| @top:???:vip | Matches the occurrence of any string that contains 1 to 3 characters | top:s:vip<br>top:su:vip<br>top:sub:vip | top:sub_env0:vip |

# 8

# Methodology Guide

This chapter contains the following sections:

- “Recommendations” : describes the complete set of recommendations to follow while developing VMM components.
- “Rules” : describes the complete set of rules to follow while developing VMM components.

## Recommendations

### Transactions

- All class properties without a rand attribute should be local when possible with the exception of constructed properties like parity etc.

- Transaction descriptors should have implementation and context references.
- All constructor arguments should have default values.
- All non-local methods should be virtual.
- Provide default constraint blocks to produce better distributions on size or duration class properties.
- Solve discriminant class properties first to avoid constraint failures.
- If the transaction object has a parent, only then you should copy the parent handle while creating a new object in the `copy()` method. Deep copy of the parent object is not recommended.
- Transactions should be factory enabled by using the `` `vmm_class_factory `` macro. You must create `copy()` and `allocate()` methods for the transactions. You can also use the shorthand macros to create the same.

## Message Service

- Issue messages of type `FAILURE_TYP` using the `` `vmm_warning() ``,`` `vmm_error() `` or`` `vmm_fatal() `` macros.
- Issue messages of type `NOTE_TYP` using the`` `vmm_note() `` macro.
- Issue messages of type `DEBUG_TYP` using the `vmm_trace()`, `` `vmm_debug() `` or `` `vmm_verbose() `` macros.
- Make calls to text output tasks only once it you have confirmed that a message is issued.

## Transactors

- You might declare transactor in a `package`.
- Transactor objects should indicate the occurrence of significant protocol and execution events via the notification service interface in the `vmm_xactor::notify` class property.
- For custom transactors modeled using `vmm_xactor`, you should ensure that `XACTOR_IDLE` and `XACTOR_BUSY` notifications are indicated or reset so that the transactor instance can appropriately agree or oppose 'end of test' completion managed through the `vmm_consensus` class.
- When you overload the `start_ph, shutdown_ph` and `reset_ph` of any transactors derived from `vmm_xactor`, you should call the `super.start_ph, super.shutdown_ph` and `super.reset_ph` so that implicit calls to `start_xactor/ stop_xactor/reset_xactor` will be made in these methods.

## Callbacks

- Transactors should call a callback method after receiving data, letting you record, modify or drop the data.
- Transactors should call a callback method before transmitting data, letting you record, modify or drop the data.
- Transactors should call a callback method after generating any new information, letting you record or modify the new information.
- Transactors should call a callback method after making a significant decision but before acting on it, letting you modify the default decision.

## Channels

- Specify channel instances as optional constructor arguments.
- Consumer transactors should use the `vmm_channel::activate()`, `vmm_channel::start()`, `vmm_channel::complete()` and `vmm_channel::remove()` methods to indicate the progress of the transaction execution.
- Indicate the `vmm_data::STARTED` and `vmm_data::ENDED` notifications if `vmm_channel::start()` and `vmm_channel::complete` are not invoked.
- Use an output “completion” channel to send back (partially) completed transactions.
- Transactors should put an incomplete transaction descriptor instance in the output channel as soon as you identify the start of a transaction.
- Requestor transactors should continue with a default response if you receive no response after the maximum allowable time interval.
- Requestor transactors should issue a warning message if you receive no response after the maximum allowable time interval.
- Transaction response request descriptors should solve to a valid random response when randomized.

## Environments

- Randomize the timing relationship of unrelated clock signals as part of the testcase configuration.

- Make a monitor transactor configurable as reactive or passive.
- The `vmm_env::cfg_dut()` method should have a fast implementation that writes to registers and memories via direct accesses.
- When an object is no longer needed, you can remove all the references to it by using `vmm_object::kill_object()`.
- Avoid creating log instances for VMM base class extensions except `vmm_data`.
- Set the parent of a VMM component either during construction or through `vmm_object::set_parent_object()`.

## Tests and Generators

- User testcases should extend `vmm_test`.
- The name of the class property containing the randomized instance should have the prefix "`randomized_`".
- You should not directly add directed stimulus to the public output channel.
- Describe exceptions separately from transactions directly in testcases.
- Use the predefined atomic generator `vmm_atomic_gen` for basic randomization.
- Use the multi-stream scenario generator for randomizing and controlling scenarios.

## Channels and TLM Ports

- Channels are preferred as input connector versus TLM interfaces. This is because, they come with a superior completion model and can feed back a status in the passed transaction directly.
- You should use VMM notification for dataless synchronization.
- For producers, use `b_transport` as they will be automatically throttled.
- For consumers, use `channel +` active slot as it provides all TLM interfaces.
- Use analysis ports for events with status/data because they are strongly-typed.

## Configuration

- For an environment, you should define and instantiate a global configuration object derived from `vmm_object`, and you should have randomizable fields.
- The global configuration object should instantiate the child config objects (which are also derived from `vmm_object`) corresponding to individual components or sub-environments (and which have been defined there).

# Rules

## Transactions

- You shall derive data and transaction model classes from the `vmm_data` class.
- All data classes shall have a public static class property referring to an instance of the message service `vmm_log`.
- All class properties corresponding to a protocol property or field shall have the `rand` attribute.
- Use a `rand` class property to define the kind of transaction you describe.
- You shall unconditionally constrain the size of a `rand` array-type class property to limit its value.
- Make all class properties with a `rand` attribute public.
- Data protection class properties shall model their validity, not their value.
- Model fixed payload data using explicit class properties.
- Use class inheritance to model different data formats, you will prefer discriminants.
- You shall not use `tagged unions` to model different data formats.
- Use a class property with the rand attribute to indicate if optional properties from different data formats are present.
- All methods shall be functions.

- Provide a virtual method to compute the correct value of each data protection class property.
- Provide a constraint block to ensure the validity of randomized class property values.
- A distribution constraint block shall constrain a single class property.
- Provide constraint blocks to avoid errors in randomized values.
- An error-prevention constraint block shall constrain a single class property.

## Message Service

- You shall issue all simulation messages through the message service.

## Transactors

- All transactor-related declarations shall have a unique prefix.
- Include all transactor-related declarations in the same file.
- implement transactors using a `vmm_xactor`.
- implement transactors in classes derived from `vmm_xactor`.
- You shall start no threads in the constructor.
- Model layers of a protocol as separate transactors.
- Identify transactors or configure as proactive, reactive or passive.

- All messages issued by a transactor instance shall use the message service interface in the `vmm_xactor::log` class property.
- Transactors shall assign the value of their `vmm_xactor::stream_id` class property to the `vmm_data::stream_id` class property of the data and transaction descriptors flowing through them.
- Transactors shall be configurable if the protocol they implement has options.
- Configure transactors using a randomizable configuration descriptor.
- Assign transactor configuration descriptor in the `configure` phase.
- Specify physical interfaces using a `virtual modport interface` and assign in the `build` phase.
- Store the `virtual interface` in a public class property.
- Command-layer transactors shall not refer directly to clock signals.
- Master transactor should be constructed with arguments allowing to be associated with its enclosing component, for example, its parent.
- Implicitly phased master transactor should implement the `connect()` phase for assigning interfaces.
- Implicitly phased master transactor should implement the `shutdown()` phase.

## Callbacks

- Transactors shall have a rich set of callback methods.
- Declare all callback methods for a transactor as virtual methods in a single class derived from `vmm_xactor_callbacks`.
- Declare callbacks as `tasks` or `void` functions.
- Arguments that you must not modify shall have the `const` attribute.
- Include a reference to the calling transactor in the callback arguments.
- Transactors shall use the ‘`vmm_callback`() macro to invoke the registered callbacks.
- You must use callback and not analysis ports to convey the transactions that may be modified.
- You must not modify transactions reported through an analysis ports.
- Transactions should be reported on analysis ports only after they have been reported on callbacks.

## Channels

- You might use a channel to exchange transactions between two transactors.
- Store references to channel instances in public class properties suffixed with “`_chan`”.

- A transactor shall not hold an internal reference to a channel instance while you stop or reset.
- Reactive and passive transactors shall allocate a new transaction descriptor instance from a factory instance using the `vmm_data::allocate()` method.
- You shall not make a transactor both a producer and a consumer for a channel instance.
- Reactive or passive transactors shall use the `vmm_channel::sneak()` method to put transaction descriptors in their output channels.
- Transactors shall clearly document the completion model input channels use.
- Reactive transactors shall clearly document the response model expected by output channels.
- Configure input channel instances with a full level of one.
- Peek transaction descriptors from the input channel.
- Remove transaction descriptors from the channel only when you complete the transaction execution.
- Use a separate channel instance for each priority or class of service.
- Consumer transactors shall use the `vmm_channel::activate()`, `vmm_channel::start()`, `vmm_channel::complete()` and `vmm_channel::remove()` methods to indicate the progress of the transaction execution.
- Consumer transactors shall use the `vmm_channel::get()` to immediately remove a transaction from the channel.

- Consumer transactors shall use the `vmm_channel::sneak()` method to add completed transaction descriptors to the completion channel.
- Producer transactors shall put transaction descriptor instances in the output channel using the `vmm_channel::sneak()` method.
- Transactors shall indicate the `vmm_data::STARTED` and `vmm_data::ENDED` notifications.
- Requestor transactors shall use the `vmm_channel::sneak()` method to post a response request into the response request channel.
- Requestor transactors shall check that a response is provided within the required time interval.
- You shall randomly generate a protocol-level response using an embedded generator.
- Protocol-level response shall be randomly generated using an embedded generator.

## Environments

- Implement the signal layer and instantiate the DUT in a top-level module.
- implement the verification environment in a top-level class.
- Declare all interface signals as `inout`.
- Sample synchronous interface signals and drive using a `clocking` block.
- Define set-up and hold time in clocking blocks using parameters.

- Specify the direction of asynchronous signals in the `modport` declaration.
- Specify the direction of synchronous signals in the `clocking` block declaration.
- Include the `clocking` block in `modports` port list instead of individual clock and synchronous signals.
- Map Signals in different `interface` instances implementing the same physical interface to each other.
- Instantiate the design and all required interfaces and signals in a `module` with no ports.
- You should add clock generation in the top-level module.
- There shall be no clock edges at time 0.
- Use the `bit` type for all clock and reset signals.
- Implement drivers and monitors as transactors.
- Transactors shall execute in the reactive region.
- Implement monitors as transactors.
- Generators shall execute in the reactive region.
- Implement generators as transactors.
- Testcases shall access elements in the top-most module or design via absolute cross-module references.
- Instantiate all transactors and generators in public class properties.
- Register first the self-checking integration callbacks with a transactor.

- Register callback extension instances that can modify or delay the transactions before the scoreboard callback extension instances.
- Register callback extension instances that do not modify the transactions after the scoreboard callback extension instances.
- An implicitly phased environment should extend from `vmm_group`.
- An explicitly phased environment should extend from `vmm_env`.
- An implicitly phased sub-environment should extend from `vmm_group`.
- An explicitly phased sub-environment should extend from `vmm_group`.
- To use explicitly phased components inside an implicitly phased environment, you should instantiate them inside a `vmm_subenv` and instantiate the `subenv` inside a implicitly phased environment.
- To use implicitly phased components inside an explicitly phased environment, you should instantiate them inside a `vmm_timeline` and instantiate the timeline inside an explicitly phased environment.

## Notifications

- Use a `vmm_notify` extension to exchange notifications between two transactors.
- Store references to notification service instances in public class properties.

- A transactor shall not hold an internal reference to a notification service instance while it is stopped or reset.

## Tests and Generators

- Design verification environments with random stimulus.
- Model a generator as a transactor.
- A generator shall have an output channel for each output stream.
- The reference to the generator output channels shall be in public class properties.
- Make optionally specifiable, a reference to pre-existing output channel instances to the generator constructor.
- A generator shall randomize a single instance located in a public class property and copy the final value to a new instance.
- Check the return value of the `randomize()` method and report an error if it is false.
- Assign the value of the `stream_id` class property of the generator to the `stream_id` class property in the randomized instance before each randomization.
- Stop generators while you inject directed stimulus.
- Generators shall provide a procedural interface to inject data or transaction descriptors.
- Use a randomized exception descriptor to randomly inject exceptions.
- An exception descriptor shall have a reference to the transaction descriptor it will be applied to.

- An exception descriptor shall have a constraint block to prevent the injection of exception by default.
- Randomize the exception descriptor using a factory pattern.
- All scenarios extended from `vmm_ms_scenario` should overload the `copy()` method. The `` `vmm_scenario_member_begin/end `` macros can be used to implement the same.
- For implicitly phased tests, you should not have any code in the build, configure and connect phases as they will not be executed when the tests are concatenated.
- You use the `` `vmm_test_concatenate() `` macros to denote whether the test can be concatenated or not.
- You should use the `` `vmm_test_concatenate `` macro to denote which phases of a timeline should roll back to for a particular test when it is concatenated.
- You should not use factory overrides in tests which will be concatenated.
- Use `configure_test_ph` for test specific code.
- For test concatenation, restore environment to original state in `cleanup_ph.`
- `Copy()` should be created for all multistream scenarios. You can use the MSS shorthand macros to create the same.

# 9

# Optimizing, Debugging and Customizing VMM

This chapter contains the following sections:


- “Optimizing VMM Components”
- “Customizing VMM”
- “Customizing VMM”

# Optimizing VMM Components

## Garbage-Collecting vmm_object Instances

Any common mistakes might contribute to the needless consumption of memory. Some basic precautions and techniques ensure that your testbench consumes as little memory as possible.

An easy way of expediting garbage collection to reduce memory usage is to turn on garbage collection for unused `vmm_object` instances. It is also possible to deallocate a complete `vmm_object` hierarchy, either from a top object or the root object.

When you no longer need an object, it is important that you remove all references to it from scoreboards and lists and to call its `vmm_object::kill_object()` method.

*Example 9-1 Killing Objects*

```
class sb;
   packet expected[$];
   ...
   function void observed(packet obs);
      packet exp = expected.pop_front();
      if (!exp.compare(obs)) ...
      exp.kill_object();
      obs.kill_object();
   endfunction
   ...
endclass
```

## Optimizing vmm_log Usage

Both `vmm_log` and `vmm_object` have names. When a class that is based on `vmm_object` also contains a `vmm_log` instance, how should you name them?

You should use the hierarchical name of the object as the instance of the `vmm_log` and you should use the name of the class as the name of the `vmm_log` instance.

This ensures that the identification of the `vmm_object` easily correlate message source to a specific object instance in the object hierarchy and render it consistent with the name of the `vmm_log` instantiated in the VMM base classes.

*Example 9-2 Associating vmm_log With Class Parent*

```
class my_class extends vmm_object;
   vmm_log log; // Not static if not too many instances

   function new(string     name   = "",
                vmm_object parent = null);
      super.new(parent, name);
      log = new("my_class",
                this.get_object_hiername());
   endfunction
   ...
endclass
```

There is no need to provide a `vmm_log` in extensions of VMM base classes as their instances already include the `vmm_log` from the base class, this applies to almost all VMM base classes. Otherwise, this results in creating two `vmm_log` instances where one is sufficient. For example, the following code creates an extra instance of the `vmm_log` class, hiding the instance already provided in the `vmm_xactor` base class:

*Example 9-3 Hiding Local vmm_log in vmm_xactor*

```
class my_xactor extends vmm_xactor;
   vmm_log log;  // Hides internal vmm_xactor::log!!

   function new(string     name   = "",
                vmm_object parent = null);
      super.new("my_xactor", name, parent);
      // Extra vmm_log instance!!
      log = new("my_xactor",
                this.get_object_hiername());
   endfunction
endclass
```

## Static vmm_log Instances

If you have a class with a large number of instances (for example, all classes extended from `vmm_data`), it is recommended that the class contain a static `vmm_log` data member.

This ensures you create only one instance of the Message Service Interface for all instances of that class.

You should initialize the static `vmm_log` instance (as well as any other static data member) by instantiating the `vmm_log` [or "static instance"] with the class declaration.

This ensures you create a single instance of the `vmm_log` class automatically during elaboration of the SystemVerilog model.

*Example 9-4 Efficient vmm_log Usage*

```
class my_data extends vmm_data;
   static vmm_log log = new("my_data", "static");
   ...
   function new(string     name   = "",
                vmm_object parent = null);
```

```
      super.new(this.log, parent, name);
      ...
   endfunction
   ...
endclass
```

If you need to initialize the static data members in the class constructor, make sure that you initialize them only once, i.e. when you create the first instance of that class.

*Example 9-5 Unique Construction of vmm_log*

```
class my_data extends vmm_data;
   static vmm_log log;
   ...
   function new(string     name   = "",
                vmm_object parent = null);
      super.new(this.log, parent, name);
      if (this.log == null) begin
         this.log = new("my_data", "static");
         super.notify.set_log(this.log);
      end
      ...
   endfunction
   ...
endclass
```

You must be careful not to allocate a `vmm_log` instance every time you create an instance of the class. This causes memory to continuously increase because you cannot garbage-collect `vmm_log` instances unless you explicitly kill them using the `vmm_log::kill()` method.

### vmm_log Instances in vmm_channel

For each `vmm_channel` instance, a `vmm_log` instance is allocated internally. This helps to debug the VMM environments. However, if there are a large number of channel instances, the additional `vmm_log` instances can lead to memory issues.

After you have debugged an environment, you need not maintain unique `vmm_log` instances for every channel as they will not issue a message.

You can thus improve the memory consumption of your environment by using a single `vmm_log` instance for all `vmm_channel` instances.

The run-time command-line option `+vmm_channel_shared_log` causes all `vmm_channel` instances in your testbench to share a single `vmm_log` instance.

## Customizing VMM

The components of VMM Standard Library are designed to meet the needs of the vast majority of users without additional customization.

However, some organizations may wish to customize the components of VMM Standard Library to offer organization-specific features and capabilities not readily available in the standard version. You should use the Standard Library customization mechanisms described in this chapter.

It is recommended to use the user-defined extension mechanisms provided by the various base and utility classes such as, virtual and callback methods.

## Adding to the Standard Library

You can extend VMM Standard Library by automatically including up to two user-specified files in the `vmm.sv` file. All user-defined customization are then embedded in the same package as the VMM Standard Library and become automatically visible without further modifications to user code.

If the define `` `VMM_PRE_INCLUDE `` is declared, the file specified by the definition is included at the beginning of the `vmm.sv` file, at the file level, before the VMM standard library package. You can use this symbol to import the pre-processor declarations needed to customize the VMM Standard Library and to define the global customization symbols.

If the define `` `VMM_POST_INCLUDE `` is declared, the file specified by the definition is included at the top of the VMM standard library package, but only after all of the known class names have been defined.

You can use this symbol to import declarations and type definitions a customized VMM Standard Library needs and the implementation of VMM Standard Library customizations that are built on the predefined classes.

*Example 9-6 Inclusion Points in the vmm.sv File*

```
`include `VMM_PRE_INCLUDE
...
package _vcs_vmm;
   typedef class vmm_xactor;
   `ifdef VMM_XACTOR_BASE
      typedef class `VMM_XACTOR_BASE
   `endif
   ...
   `include `VMM_POST_INCLUDE
   ...
```

```
   class vmm_broadcast extends `VMM_XACTOR;
   ...
endpackage
```

Note: The symbol definition must include the double quotes surrounding the filename.

*Example 9-7 Adding to VMM Standard Library*

```
vcs -sverilog -ntb_opts rvm \
   +define+VMM_PRE_INCLUDE=\"vmm_defines.svh\" \
   +define+VMM_POST_INCLUDE=\"acme_stdlib.sv\" ...
```

## Customizing Base Classes

The `vmm_data`, `vmm_channel`, `vmm_xactor` and `vmm_env` base classes are designed to be specialized into different protocol-specific transaction descriptors, transactors and verification environments.

You can create a set of organization-specific base classes to introduce organization-specific generic functionality to all VMM components that organization creates, as shown in Example 9-8 and Example 9-9.

*Example 9-8 Organization-Specific Transactor Base Class*

```
class acme_xactor extends vmm_xactor;
   ...
endclass: acme_xactor
```

*Example 9-9 Transactor Based on Organization-Specific Base Class*

```
class ahb_master extends acme_xactor;
   ...
endclass: ahb_master
```

A problem exists that any VMM component not written by the organization, such as the one shown in Example 9-10, will not be based on that organization's base class. This makes several kinds of features (such as automatically starting all transactor instances when `acme_env::start()` is executed) impossible to create.

*Example 9-10 Transactor Based on Standard Base Class*

```
class ocp_master extends vmm_xactor;
   ...
endclass: ocp_master
```

You can use the following techniques to customize the VMM base classes. Although you describe the techniques using the `vmm_xactor` base class, you can apply the same techniques to the `vmm_data` and `vmm_env` base classes as well.

The only difference is that their respective symbols would start with `"VMM_DATA"` and `"VMM_ENV"` respectively, instead of `"VMM_XACTOR"`.

"Customizing VMM" on page 6 details the customization macros available with all predefined components in the VMM standard library.

## Symbolic Base Class

All VMM-compliant components are based on the symbolic base class specified by the `'VMM_XACTOR` symbol, as shown in Example 9-11. Upon compilation, you can redefine the symbol (defined by default to be `"vmm_xactor"`) to cause the transactor to be based on an alternate (but homomorphic) base class, as shown in Example 9-12. You should ultimately base this alternate base class on `vmm_xactor`.

*Example 9-11 Transactors Based on Symbolic Base Class*

```
class ahb_master extends `VMM_XACTOR;
   ...
endclass: ahb_master

class ocp_master extends `VMM_XACTOR;
   ...
endclass: ocp_master
```

*Example 9-12 Redefining the Symbolic vmm_xactor Base Class*

```
`define VMM_XACTOR acme_xactor
```

In the above example, the simple mechanism works if the constructor of the alternate base class has the exact same arguments as the `vmm_xactor` base class. Additional macros are provided to support non-homomorphic constructors.

You should create the transactors using (see Example 9-13),

- VMM_XACTOR_NEW_ARGS
- VMM_XACTOR_NEW_CALL

Note: A comma does not precede either of the macros. The purpose of this is to handle any instance where you do not define the symbols. It also implies that whenever you define these symbols, their definition must start with a comma.

*Example 9-13 Transactor Supporting Non-Homomorphic Base Constructor*

```
class ocp_master extends `VMM_XACTOR;
   ...
   extern function new(string inst,
                       int    stream_id = -1
                       `VMM_XACTOR_NEW_ARGS);
   ...
endclass: ocp_master

function ocp_master::new(string inst,
```

```
                              int    stream_id
                              `VMM_XACTOR_NEW_ARGS);
   super.new("OCP Master", inst, stream_id
`VMM_XACTOR_NEW_CALL);
   ...
endfunction: new
```

You can then use an alternate transactor base class by defining the symbolic constructor argument macros appropriately. Example 9-14 shows how to use the alternate base class shown in Example 9-15.

*Example 9-14 Using a Non-Homomorphic Transactor Base Class*

```
`define VMM_XACTOR             acme_xactor
`define VMM_XACTOR_NEW_ARGS  , acme_xactor parent = null, \
                               int key = -1
`define VMM_XACTOR_NEW_CALL , parent, key
```

In order to be backward compatible with existing VMM-compliant transactors, the first arguments of the alternate base class must match the arguments of the standard `vmm_xactor` base class and provide default argument values for any subsequent arguments, as shown in Example 9-15.

*Example 9-15 Backward-Compatible Alternate Base Class*

```
class acme_xactor extends vmm_xactor;
   function new(vmm_object parent   = null,
                string      name      = "",
                string      inst      = "",
                int         stream_id = -1,
                acme_xactor parent    = null
                int         key       = -1);
      super.new(name, inst, stream_id);
      super.set_parent_object(parent);
      ...
   endfunction: new
endclass: acme_xactor
```

You can write all predefined transactions, transactors and verification environments in the VMM library (`vmm_broadcast`, `vmm_scheduler`, `vmm_atomic_gen` and `vmm_scenario_gen`) and application packages (`vmm_rw_access`, `vmm_rw_xactor`, `vmm_ral_env`) using symbolic base classes and additional constructor arguments. By default, you base them on the standard VMM base classes.

"Customizing VMM" on page 6 details the customization macros available with all predefined components in the VMM standard library. For details, see User's Guide that corresponds to VMM application package for the available customization macros.

It is important to note that the implementation of virtual methods sometimes needs to invoke the base class implementation (for example, `vmm_xactor::start_xactor()`) and sometimes might not (for example, `vmm_data::compare()`). When using an alternate `vmm_data` base class, it is important to understand that except for `vmm_data:copy_data()`, their respective extensions call none of the virtual methods in the base class.

## Customizing Utility Classes

The `vmm_log`, `vmm_notify` and `vmm_consensus` utility classes are designed to be used as-is when creating verification components, verification environments and test cases. You can create a set of organization-specific utility classes to introduce organization-specific generic functionality to all VMM components, environments and test cases created by that organization, as shown in Example 9-16.

*Example 9-16 Organization-Specific Message Interface*

```
class acme_log extends vmm_log;
```

```
   ...
endclass: acme_log
```

A problem exists that any VMM component not written by the organization, such as the standard library component shown in Example 9-17, will not use that organization's utility class. This makes several kinds of features impossible to create.

*Example 9-17 VMM Base Class Using Standard Utility Class*

```
class vmm_xactor;
   vmm_log log;
   ...
endclass: vmm_xactor
```

You can use the following techniques to customize the VMM utility classes. Although the techniques are described using the `vmm_log` utility class, you can apply them to the `vmm_notify` and `vmm_consensus` utility classes as well. The only difference is that their respective symbols would start with `"VMM_NOTIFY"` and `"VMM_CONSENSUS"` respectively instead of `"VMM_LOG"`.

"Customizing VMM" on page 6 details the customization macros available with all predefined components in VMM standard library.

## Symbolic Utility Class

All VMM-compliant components should use the symbolic base class specified by the `` `VMM_LOG `` symbol, as shown in Example 9-18 and Example 9-19. You can redefine the symbol (defined by default to be `"vmm_log"`) at compile-time to cause the base classes and components to use an alternate (but homomorphic) utility class, as shown in Example 9-20. This alternate utility class should ultimately be based on `vmm_log`.

*Example 9-18 VMM Base Class Using Symbolic Utility Class*

```
class vmm_xactor;
   `VMM_LOG log;
   ...
endclass: vmm_xactor
```

*Example 9-19 Scoreboard Using Symbolic Utility Class*

```
class scoreboard;
   `VMM_LOG log;
   ...
endclass: scoreboard
```

*Example 9-20 Redefining the Symbolic vmm_log Utility Class*

```
`define VMM_LOG acme_log
```

The simple mechanism shown above works if the constructor of the alternate utility class has the exact same arguments as the `vmm_log` utility class.

All predefined elements in the VMM library and application packages are written using symbolic utility classes. By default, they use the standard VMM utility classes.

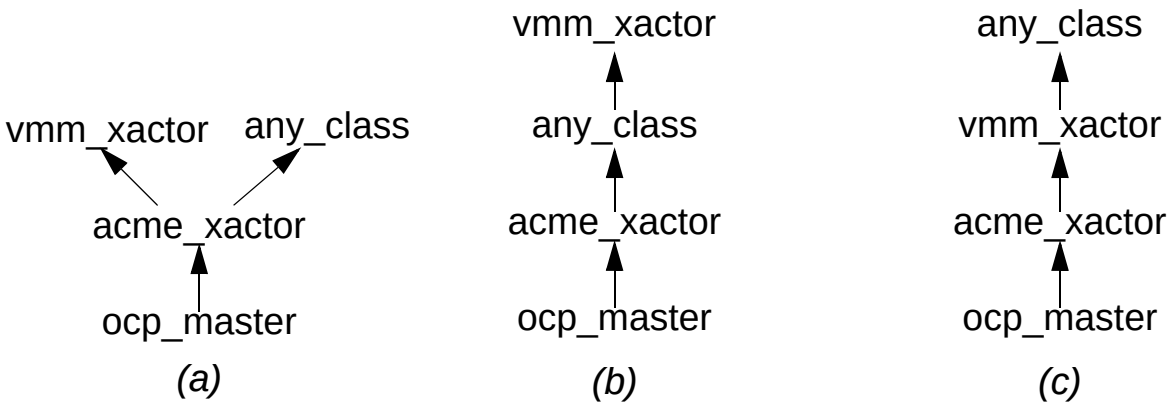
"Customizing VMM" on page 6 details the customization macros available with all predefined components in the VMM standard library. See the User's Guide which corresponds to the appropriate VMM application package for the available customization macros.

## Underpinning Classes

SystemVerilog does not support multiple inheritance. You should limit class inheritance to a single lineage. It might be desirable to have all transactors derived from more than one base class.

For example, it might be useful to have all transactors derived from the organization-specific transactor base class and the organization-specific `"any class"` base class. Figure 9-1(a) displays how to accomplish this in a language supporting multiple inheritance such as, C++. Figure 9-1(b) and Figure 9-1(c) show two alternative implementations in a single-inheritance language such as, SystemVerilog.

*Figure 9-1 Transactor Inheriting From More Than One Class*




You can implement the inheritance shown in Figure 9-1(b) by using the `VMM_XACTOR` symbolic base class macros described in “Customizing Base Classes” on page 8. However, you can do this if you can in turn base the ultimate base class on the `vmm_xactor` base class-which is not always possible or sensible.

It is possible to base the VMM Standard Library base and utility classes on a suitable user-defined base class. Although these techniques are described for the `vmm_xactor` base class, you can apply them to the all other base and utility classes defined in the VMM Standard Library as well. The only difference is that their respective symbols would start, for example, with "`VMM_DATA`" and "`VMM_LOG`" respectively instead of "`VMM_XACTOR`".

Note: You can use the `+define+VMM_11` VCS compilation option to avoid the potential conflicts that might be introduced while underpinning base classes in VMM D-2010.06 and later versions. This is because many new VMM1.2 functionality is introduced through the same mechanism of underpinning base classes.

Any Standard Library base or utility class can be based on a user-defined class by appropriately defining the following macros:

- VMM_XACTOR_BASE
- VMM_XACTOR_BASE_NEW_ARGS
- VMM_XACTOR_BASE_NEW_CALL
- VMM_XACTOR_BASE_METHODS

If you define the VMM_XACTOR_BASE macro, the `vmm_xactor` base class becomes implemented as shown in Example 9-21.

*Example 9-21 Targetable vmm_xactor Base Class*

```
class vmm_xactor extends `VMM_XACTOR_BASE;
   ...
   function new(string name,
                string inst,
                int    stream_id = -1
                `VMM_XACTOR_BASE_NEW_ARGS);
`ifdef VMM_XACTOR_BASE_NEW_CALL
      super.new(`VMM_XACTOR_BASE_NEW_CALL);
`endif
      ...
   endfunction: new

   `VMM_XACTOR_BASE_METHODS

   ...
endclass: vmm_xactor
```

Example 9-22 shows how the `vmm_xactor` base class can be targeted to the base class shown in Example 9-23.

*Example 9-22 Underpinning vmm_xactor Base Class*

```
‘define VMM_XACTOR_BASE                    any_class
‘define VMM_XACTOR_BASE_METHODS \
    virtual function string whoami(); \
       return "vmm_xactor"; \
    endfunction: whoami
```

*Example 9-23 Ultimate Base Class*

```
virtual class any_class;
   virtual function string whoami();
endclass: any_class
```

If you choose to expose the arguments of the new base class underpinning the `vmm_xactor` base class to the transactors, you must add the content of the following symbols:

- VMM_XACTOR_BASE_NEW_ARGS
- VMM_XACTOR_BASE_NEW_CALL

...to the following symbols, respectively:

- VMM_XACTOR_NEW_ARGS
- VMM_XACTOR_NEW_CALL

## Base Classes as IP

You can apply the base class underpinning mechanism shown prior recursively to any class hierarchy. This allows the creation of base class IP that you can position between the inheritances of two appropriately written classes.

Example 9-24 shows a VMM-compliant transactor base class provided by company XYZ. Any organization, whose transactor base class has a structure similar to this one can then leverage that base class by inserting it into their transactor class hierarchy.

*Example 9-24 Transactor Base Class IP*

```
`include "vmm.sv"
`ifndef XYZ_XACTOR_BASE
  `define XYZ_XACTOR_BASE          `VMM_XACTOR
`endif
`ifndef XYZ_XACTOR_BASE_NEW_ARGS
  `define XYZ_XACTOR_BASE_NEW_ARGS `VMM_XACTOR_NEW_ARGS
  `define XYZ_XACTOR_BASE_NEW_CALL `VMM_XACTOR_NEW_CALL
`endif

class xyz_xactor extends XYZ_XACTOR_BASE;
   ...
   function new(string      name,
                string      inst,
                int         stream_id = -1,
                bit         foo       = 0
                `XYZ_XACTOR_BASE_NEW_ARGS);
      super.new(name, inst, stream_id
`XYZ_XACTOR_BASE_NEW_CALL);
      ...
   endfunction: new
   ...
endclass: xyz_xactor
```

By default, this third-party base class should extend the `vmm_xactor` base class and can thus be easily inserted between the organization's transactor base class and the `vmm_xactor` base class as shown in Example 9-25. But it can also be inserted above the organization's own transactor base class as shown in Example 9-26.

*Example 9-25 Using Base Class IP Below Organization Base Class*

```
`define ACME_XACTOR_BASE           xyz_xactor
`define ACME_XACTOR_BASE_NEW_ARGS  , bit foo = 0 \
                                  `XYZ_XACTOR_BASE_NEW_ARGS
```

```
`define ACME_XACTOR_BASE_NEW_CALL
```

*Example 9-26 Using Base Class IP Above Organization Base Class*

```
`define XYZ_XACTOR_BASE              acme_base
`define XYZ_XACTOR_BASE_NEW_ARGS    , acme_xactor parent =
null, \
                                      int key = -1
`define XYZ_XACTOR_BASE_NEW_CALL    , parent, key

`define VMM_XACTOR                   xyz_xactor
`define VMM_XACTOR_NEW_ARGS        , bit foo = 0 \
                                   `XYZ_XACTOR_BASE_NEW_ARGS
`define VMM_XACTOR_NEW_CALL        `XYZ_XACTOR_BASE_NEW_CALL
```

# 10

# Primers

This chapter contains the following sections:


- Multi-Stream Scenario Generator Primer
- Class Factory Service Primer
- Hierarchical Configuration Primer
- RTL Configuration Primer
- Implicitly Phased Master Transactor Primer

# Multi-Stream Scenario Generator Primer

## Introduction

Multi-Stream Scenario Generator (MSSG) provides an efficient way of generating and synchronizing stimulus to various BFMs. This helps you in reusing block-level scenarios in subsystem and system levels and controlling and synchronizing the execution of those scenarios of same or different streams. Single stream scenarios can also be reused in multi-stream scenarios.

`vmm_ms_scenario` and `vmm_ms_scenario_gen` are the base classes provided for this functionality. This section describes the various types of usage of multi-stream scenario generation with these base classes.

Multi-Stream Scenario (MSS) extend `vmm_ms_scenario` class and define the execution of the scenario `execute()` method. By controlling the content of `execute()` method entirely, you can execute single or multiple, transactions or scenarios.

Execution can be single-threaded, multi-threaded, reactive, etc. depending on your requirement. Then the scenario object has to be registered with a multi-stream generator. MSSG executes the registered scenario. Multiple MSS can be registered to the same MSSG.

The following sections explain how to implement Multi-stream scenarios and create hierarchical scenarios using a procedural approach:

## Step1: Creation of Scenario Class

You can create a scenario class by extending `vmm_ms_scenario` and defining any properties as rand if they are intended to be randomized before the execution of the `execute()` method. Implement `copy()` method by copying the contents of the scenario. This can be done by using `` `vmm_data_scenario* `` macros also.You then define the `execute()` method according to the need.

You can update `'n',` the argument of `execute` method to keep track of the number of transactions executed by the generator to which this scenario is registered. Number of transactions is controlled by `stop_after_n_insts` property of the generator.

It is required that for each scenario the `vmm_ms_scenario::copy()` should be overloaded for multistream scenarios to return the copy of the scenario.

The easiest way to achieve this is to use the shorthand macros.

```
`vmm_scenario_member_begin(..)
...
vmm_scenario_member_end(..)
```

Note: These macros create a default constructor. If there is a need to create your own constructor, you need to explicitly define the macro, `` `vmm_scenario_new(..) `` in addition to the above macros.

*Example 10-1 Creating Basic MSS*

```
class my_scenario extends vmm_ms_scenario;
    // NUM will be randomized before execute() is called

  rand int NUM;
```

```
  //Implementing copy() and application methods through
    macro
`vmm_scenario_new(my_scenario)
`vmm_scenario_member_begin(my_scenario)
`vmm_scenario_member_int(NUM, DO_ALL)
`vmm_scenario_member_end(my_scenario)
  constraint cst_num {
    NUM inside {[1:10]};
  }
 `vmm_scenario_member_begin(my_scenario)
      `vmm_scenario_member_scalar(SCN_KIND, DO_ALL)
    `vmm_scenario_member_end(my_scenario)

  function new(vmm_ms_scenario parent = null);
    super.new(parent);
    trans = new();
  endfunction

  task execute(ref int n);
    for (int i=0; i<NUM; i++) begin
      $display("This is a dummy transaction: %0d", n);
      n++;  // Incrementing n after execution

          // of every transaction

    end
  endtask
endclass
```

## Step 2: Usage of Logical Channels in MSS

Usually, scenarios are in charge of putting transactions to VMM channels. To facilitate this, `vmm_channel` instances can be registered to the MSSG by name and be accessed in the `vmm_ms_scenario` through its `get_channel()` method by specifying the same name.

You must declare the `vmm_channel` instances in the top environment and then associate it the `register_channel()` method.

*Example 10-2 Using Logical vmm_channel in MSS*

```
class my_scenario extends vmm_ms_scenario;
  ….
  task execute(ref int n);
    my_trans tr = new;
    // Returns channel with name “MY_CHAN” if
    // available in the generator where this scenario
    // is registered
    vmm_channel my_chan = get_channel(“MY_CHAN”);
    tr.randomize();
    my_chan.put(tr);
    tr.notify.wait_for(vmm_data::ENDED);
  endtask
endclass

my_scenario scn = new();
my_trans_channel chan = new(“mychan”, “mychaninst”);
vmm_ms_scenario_gen gen = new(“GEN”);

// registering vmm_channel by name "MY_CHAN"
gen.register_channel(“MY_CHAN”, chan);
```

## Step 3: Registration of MSS in MSSG

You can instantiate the previous MSS and register it to MSSG object through `register_ms_scenario()` method.

MSSG randomizes the scenario and calls its `execute()` method.

Any number of MSS can be registered to a MSSG by default. It executes scenarios in round robin order until the `stop_after_n_scenarios or stop_after_n_insts` limit is reached.

*Example 10-3 Registration of MSS in MSSG*

```
my_scenario scn = new();
vmm_ms_scenario_gen gen = new("GEN");
gen.register_ms_scenario("MY_SCN", scn);

// Generator randomizes and executes the registered
// scenario 5 times.
gen.stop_after_n_scnearios = 5;
```

## Complete Example of a Simple MSSG

The following code snippets show the basic usage of MSS where two different types of transactions are executed concurrently.

You can change the order of execution according to the requirement (dynamic, reactive, etc.).

Example 10-4 shows how to model an ALU and a APB transaction.

*Example 10-4 Implementation of ALU and APB Transactions*

```
//ALU transaction
class alu_trans extends vmm_data;
  typedef enum bit [2:0] {ADD=3'b000, SUB=3'b001,
                          MUL=3'b010, LS=3'b011,
                          RS=3'b100} kind_t;
  rand kind_t kind;
  rand bit [3:0] a, b;
  rand bit [6:0] y;

  `vmm_data_member_begin(alu_trans)
    `vmm_data_member_enum(kind, DO_ALL)
    `vmm_data_member_scalar(a, DO_ALL)
    `vmm_data_member_scalar(b, DO_ALL)
    `vmm_data_member_scalar(y, DO_ALL)
  `vmm_data_member_end(alu_trans)
endclass
`vmm_channel(alu_trans)
```

```
//APB transaction
class apb_trans extends vmm_data;
  typedef enum bit {READ=1'b0, WRITE=1'b1} kind_e;
  rand bit [31:0] addr;
  rand bit [31:0] data;
  rand kind_e kind;
  `vmm_data_member_begin(apb_trans)
    `vmm_data_member_scalar(addr, DO_ALL)
    `vmm_data_member_scalar(data, DO_ALL)
    `vmm_data_member_enum(kind, DO_ALL)
  `vmm_data_member_end(apb_trans)
endclass
`vmm_channel(apb_trans)
```

Example 10-5 shows how to create a scenario that randomizes a stream of ALU transactions and a stream of APB transactions.

Note: Each randomized transaction is posted to its respective logical channel.

*Example 10-5 Implementation of MSS to Randomized ALU and CPU Transactions*

```
// Multi stream scenario with concurrent execution of 2
// transactions of different streams
class my_scenario extends vmm_ms_scenario;
  alu_trans_channel alu_chan;
  apb_trans_channel apb_chan;

  //Transaction gets randomized when this
  // ms scenario gets randomized
  rand apb_trans apb_tr;

  alu_trans alu_tr; //Transaction won't get randomized
  int MY_SCN = define_scenario("MY_SCN", 0);
  function new(vmm_ms_scenario parent=null);
    super.new(parent);
    apb_tr = new();
  endfunction

  virtual task execute(ref int n);
```

```
    $cast(alu_chan, this.get_channel("ALU_SCN_CHAN"));
    $cast(apb_chan, this.get_channel("APB_SCN_CHAN"));
    fork
    begin // Randomize ALU transaction
      alu_trans tr;
      $cast(tr, alu_tr.copy());
      tr.randomize();
      alu_chan.put(tr);
     n++; //User must update the number of transactions
    end
    begin  // Randomize APB transaction
      apb_trans tr;
      $cast(tr, apb_tr.copy());
      apb_chan.put(tr);
     n++; //User must update the number of transactions.
    end
    join
  endtask
endclass
```

Example 10-6 shows how to extract transactions from two registered logical channels that are attached to the previously declared MSSG. A simple code block gets each transaction stream. This example outputs APB transactions and ALU transactions concurrently five times as the scenario gets executed five times (`stop_after_n_scenarios = 5`).

*Example 10-6 Implementation of MSSG*

```
program automatic P;

initial begin
  alu_trans_channel alu_chan = new("ALU_CHAN", "Chan");
  apb_trans_channel apb_chan = new("APB_CHAN", "Chan");
  vmm_ms_scenario_gen gen = new("Gen"); //MSSG
  my_scenario scn = new;

  // Register alu_chan channel to the generator
  gen.register_channel("ALU_SCN_CHAN", alu_chan);

  //register apb_chan channel to the generator
```

```
  gen.register_channel("APB_SCN_CHAN", apb_chan);

  // register multi stream scenario to the generator
  gen.register_ms_scenario("SCN", scn);

  gen.stop_after_n_scenarios = 5;
  gen.start_xactor();

  fork
    repeat(5) begin
      alu_trans tr;
      alu_chan.get(tr);
      tr.display("ALU:");
    end
   repeat(5) begin
     apb_trans tr;
     apb_chan.get(tr);
     tr.display("APB:");
   end
  join
end

endprogram
```

# Class Factory Service Primer

## Introduction

In a typical verification environment, there are generators that create transactions or transaction scenarios. And there are transactors that process these transactions and transmit them to the design under test (DUT). The transactions and scenarios are modeled as classes with built-in members, methods and constraints.

It is often required for the tests to provide additional features such as different constraints to target various design functionalities to verify. Class Factory service provides an easy way to construct any kind of object such as, transaction or scenario. This factory can be overridden by a similar one without having to modify the generators and transactors it belongs to.

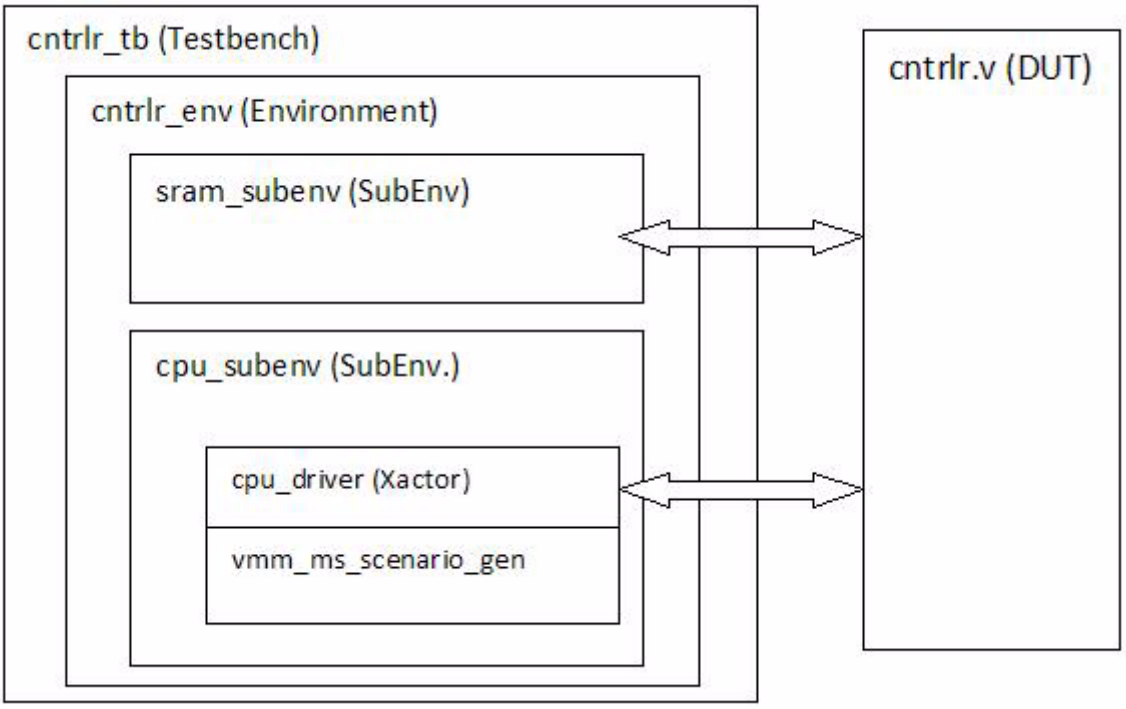
This section explains how to construct and override objects by using the class factory service.

The example for design under test (DUT) contains two functional interfaces, the CPU interface and the SRAM interface. The top-level testbench instantiates a CPU subenvironment and a SRAM subenvironment.

There are two transactors inside the CPU subenvironment:

- CPU driver that is responsible for driving and sampling the DUT signals through the interface.
- VMM multi-stream scenario generator (MSSG) that is responsible for creating CPU transaction scenarios for the CPU driver to process.

*Figure 10-1 Environment Block Diagram*




## Step 1: Modeling Classes to be Factory Ready

This step explains how to model transactions and scenarios to be factory ready.

To create all underlying factory infrastructures and to make a transaction factory ready, you use the `` `vmm_class_factory() `` macro with the name of the class as its argument.

It is important that you provide a `new(), allocate()` and `copy()` method in the transaction or any other component which is desired to be made factory ready.

The use of `vmm_class_factory macro with the specified data type creates four class methods, which are static to ensure that they can be called from anywhere:

- classname::this_type() returns the handle to the class factory.
- classname::create_instance() constructs an instance of the specified class type. This method is similar to classname::new() but should be used to ensure the constructed object can be replaced.
- classname::override_with_new() replaces the matching class instance by the specified class. This method uses the classname::allocate() method of the specified class to create a new instance.
- classname::override_with_copy() replaces the matching class instance by a copy of the provided class instance. This method uses the classname::copy() method of the specified class to create a new instance.

The cpu_trans class describes the properties of a CPU transaction. This class is extended from the vmm_data base class and uses the `vmm_data_member_* shorthand macros to implement all the virtual methods including vmm_data::copy(), and vmm_data::allocate().

These two methods must be overridden with a transaction-specific implementation as they are used when a factory is replaced in the environment and/or the tests, which is described in the later sections.

*Example 10-7 Modeling Factory-Enabled Transaction*

```
class cpu_trans extends vmm_data;
   typedef enum bit {READ = 1'b1, WRITE =  1'b0} kind_e;
```

```
    rand bit [7:0] address;
    rand bit [7:0] data;
    rand kind_e    kind;

    `vmm_data_member_begin(cpu_trans)
      `vmm_data_member_scalar(address, DO_ALL)
      `vmm_data_member_scalar(data, DO_ALL)
      `vmm_data_member_enum(kind, DO_ALL)
    `vmm_data_member_end(cpu_trans)

    `vmm_class_factory(cpu_trans)

endclass
```

To build a multi-stream scenario (MSS) based on the `cpu_trans` objects, a `cpu_rand_scenario` class is extended from the `vmm_ms_scenario` base class.

VMM provides several useful shorthand macros such as `` `vmm_scenario_member_*() `` to implement all the virtual methods in the `vmm_ms_scenario` base class.

As described earlier, it is important to implement its `allocate()` and `copy()` methods. This is automatically done by using the `` `vmm_class_factory `` macro with the class name as shown in Example 10-8 to make the transaction scenario class factory ready.

*Example 10-8 Modeling Factory-Enabled Scenario*

```
class cpu_rand_scenario extends vmm_ms_scenario;
  cpu_trans       blueprint;
 `vmm_scenario_new(cpu_rand_scenario)

 `vmm_scenario_member_begin(cpu_rand_scenario)
   `vmm_scenario_member_vmm_data(blueprint, DO_ALL,
DO_REFCOPY)
 `vmm_scenario_member_end(cpu_rand_scenario)
  ...
```

```
  `vmm_class_factory(cpu_rand_scenario)
endclass
```

## Step 2: Instantiating a Factory in Transactor

In the previous step, both the `cpu_trans` transaction class and the `cpu_rand_scenario` MSS have been made factory ready by using the `` `vmm_class_factory() `` macro in the respective classes.

In this step, you will learn how to instantiate a transaction factory in a transactor. A typical situation is to instantiate a transaction factory in MSS.

The `cpu_trans` transaction is instantiated as a blueprint object for the `cpu_rand_scenario` MSS in its constructor. Keeping the instantiation and the randomization of the `cpu_trans` objects separately is necessary so that the factory can be replaced anywhere in the verification environment and tests before the `cpu_trans` objects get randomized in the `cpu_rand_scenario::execute()` method.

*Example 10-9 Instantiating a Transaction in MSS*

```
class cpu_rand_scenario extends vmm_ms_scenario;
  cpu_trans       blueprint;
  function new();
     // Construct the blueprint with name "blueprint"
     // and provides a handle to cpu_rand_scenario
     blueprint = cpu_trans::create_instance(this,
                    "blueprint", `__FILE__, `__LINE__);
  endfunction
  virtual task execute(ref int n);
     cpu_trans tr;
     bit res;
     vmm_channel chan;
     if (chan == null) chan = get_channel("cpu_chan");
```

```
      $cast(tr, blueprint.copy());
      res = tr.randomize();
      chan.put(tr);
    endtask
    `vmm_class_factory(cpu_rand_scenario)
  endclass
```

In the `execute()` method of the MSS, the blueprint is first copied and then cast to the local variable `tr` of the `cpu_trans` type. Casting is necessary because the blueprint is the object that extends from `cpu_trans` and contains the user-defined features such as, additional members and constraints but the object to randomize is of the `cpu_trans` type.

A new instance of the blueprint is required before each randomization as the channel stores the `cpu_trans` objects by reference and not by copy.

## Step 3: Instantiating a MSS Factory in MSSG

In the previous step, the `cpu_rand_scenario` MSS class has been made factory ready by using the `` `vmm_class_factory() `` macro.

In this step, you will learn how to instantiate MSS and MSSG.

The MSS and MSSG are instantiated in the CPU subenvironment class, `cpu_subenv`. Testbench components such as MSSG are constructed in the `build_ph()` phase in the pre-test timeline.

The creation of the `cpu_rand_scenario` instance and its registration to MSSG using `vmm_ms_scenario_gen::register_ms_scenario()` are

done in the `start_of_sim_ph()` phase in the test timeline, so it allows the tests to override the factory in the `configure_test_ph()` phase, the first in the test timeline.

*Example 10-10 Instantiating MSS in MSSG*

```
class cpu_subenv extends vmm_unit;
  `vmm_typename(cpu_subenv)
  ...
  vmm_ms_scenario_gen       gen;
  cpu_rand_scenario         rand_scn;

  function void build_ph();
     ...
    this.gen = new({get_object_name(), "Gen"},0, this);
  endfunction

  function void start_of_sim_ph();
     // Construct the scenario blueprint with
     // name "rand_scn" and provides a handle
     // to cpu_subenv
    rand_scn = cpu_rand_scenario::create_instance(this,
                      "rand_scn", `__FILE__, `__LINE__);
   this.gen.register_ms_scenario("rand_scn", rand_scn);
      ...
  endfunction

endclass
```

## Step 4: Replacing a Factory

In the previous step, the `cpu_rand_scenario` factory was instantiated in the CPU environment MSSG.

In this step, you will learn how to replace it, either by a copy or by new MSS.

After you have created the MSS in the `build_ph()` phase of the CPU subenvironment, it is ready to be replaced using `override_with_new()` and `override_with_copy()` methods, as shown in the examples below.

For an object that needs to be replaced by an extended object with added constraints and/or data members, you should use the `override_with_new()` method.

For an object that needs to be replaced by a similar object but with different data member values, you should use the `override_with_copy()` method. These examples also demonstrate how both the transaction and the transaction scenario factories can be replaced.

## Step 4a: Replacing a Factory by a New One

A MSS is derived from `cpu_rand_scenario`. It might include other MSS and other properties and constraints. This extended scenario can be used to override the `cpu_rand_scenario` factory in the CPU subenvironment with the static method `cpu_rand_scenario::override_with_new()`.

It is important that you implement the `allocate()` and `copy()` methods of this new scenario. You can do this automatically by using the shorthand macros, `vmm_scenario_member_*`.

Example 10-11 demonstrates the usage for factory for the multi-stream scenario. Recall that the `cpu_rand_scenario` factory instance is created in the `start_of_sim_ph()` phase of the test timeline. In order to override the factory, the `override_with_*()` methods must be called before the factory instance creation. The phase before the `start_of_sim_ph()` in the same test timeline is

the `configure_test_ph()` phase and this is where `cpu_rand_scenario::override_with_new()` is called in the example.

*Example 10-11 Implementing a New MSS and Instantiating It*

```
class my_cpu_mss extends cpu_rand_scenario;
  cpu_write_scenario  write_scn;
  cpu_read_scenario   read_scn;
  rand bit [31:0] Addr;
  rand bit [7:0] Data;
  `vmm_scenario_new (my_cpu_mss)

`vmm_scenario_member_begin(my_cpu_mss)
   `vmm_scenario_member_vmm_scenario(write_scn, DO_ALL)
  `vmm_scenario_member_vmm_scenario(read_scn, DO_ALL)
  `vmm_scenario_member_scalar(Addr, DO_ALL)
  `vmm_scenario_member_scalar(Data, DO_ALL)
`vmm_scenario_member_end(my_cpu_mss)
  ...
  virtual task execute(ref int n);
    cpu_trans tr;
    bit res;
    vmm_channel chan;
    write_scn.randomize() with {
      addr == this.Addr; data == this.Data;
    };
    write_scn.execute(n);
    read_scn.randomize() with {
      addr == this.Addr;
    };
    read_scn.execute(n);
  endtask
endclass

class test_write_read_same_addr extends vmm_test;
   // this macro defines a get_typename function
   // that returns the handle to the current object
  `vmm_typename(test_write_read_same_addr)

  function new(string name);
```

```
    super.new(name);
  endfunction


  virtual function void configure_test_ph();
    // Replace matching MSS blueprint called
    //  "*:CPU:rand_scn"
    // with the extended MSS my_cpu_mss
    cpu_rand_scenario::override_with_new(
                       "@%*:CPU:rand_scn",
                       my_cpu_mss::this_type(),
                       log, `__FILE__, `__LINE__);
  endfunction
endclass
```

After the `test_write_read_same_addr` test class is defined, it is instantiated with the standard call to its constructor, `vmm_test::new()`. During the construction, the test object is added to the test registry in the `vmm_simulation`. When the `vmm_simulation` reaches its test timeline, the test is run when selected by `+vmm_test` at VCS runtime.

## Step 4b: Replacing a Factory by a Copy

Similarly, the underlying `cpu_trans` transaction factory can also be replaced.

Example 10-12 demonstrates the usage for factory for the transaction. To override the factory in this example, you must call the `cpu_trans::override_with_copy()` static method before you call the `static method cpu_trans::create_instance()` static method to create the factory instance of the blueprint.

The blueprint instance is created in the construction of the `cpu_rand_scenario` which is instanced in the `start_of_sim_ph()` phase of the CPU subenvironment. Consequently, if you call the `cpu_trans::override_with_copy()` method in the `configure_test_ph()` phase of the test timeline, the `cpu_trans` factory is successfully overridden.

*Example 10-12 Implementing a Copy Transaction and Instantiating It*

```
class test_read_back2back extends vmm_test;
  function new(string name);
     super.new(name);
  endfunction

  virtual function void configure_test_ph();
     cpu_trans tr = new();
     // modify the content of the extended cpu_trans
     tr.address = 'habcd_1234;
     // turn off its randomization
     tr.address.rand_mode(0);

     // replace all matching cpu_trans blueprint with tr
     cpu_trans::override_with_copy("@%*", tr,
                                   log, `__FILE__, `__LINE__);
  endfunction
endclass
```

## Summary

VMM provides a pre-defined factory API that greatly eases the implementation of object factories.

You should perform the following three steps to create an effective factory service:

1. Model the object using a class and use the `vmm_class_factory macro to make the object factory enabled.
2. In testbench components that create instances of the objects, create the object instance using the object's create_instance() method so that this object instance can be replaced easily, as required.
3. Replace the object factory as needed using either the override_with_copy() or override_with_new() methods of the object.

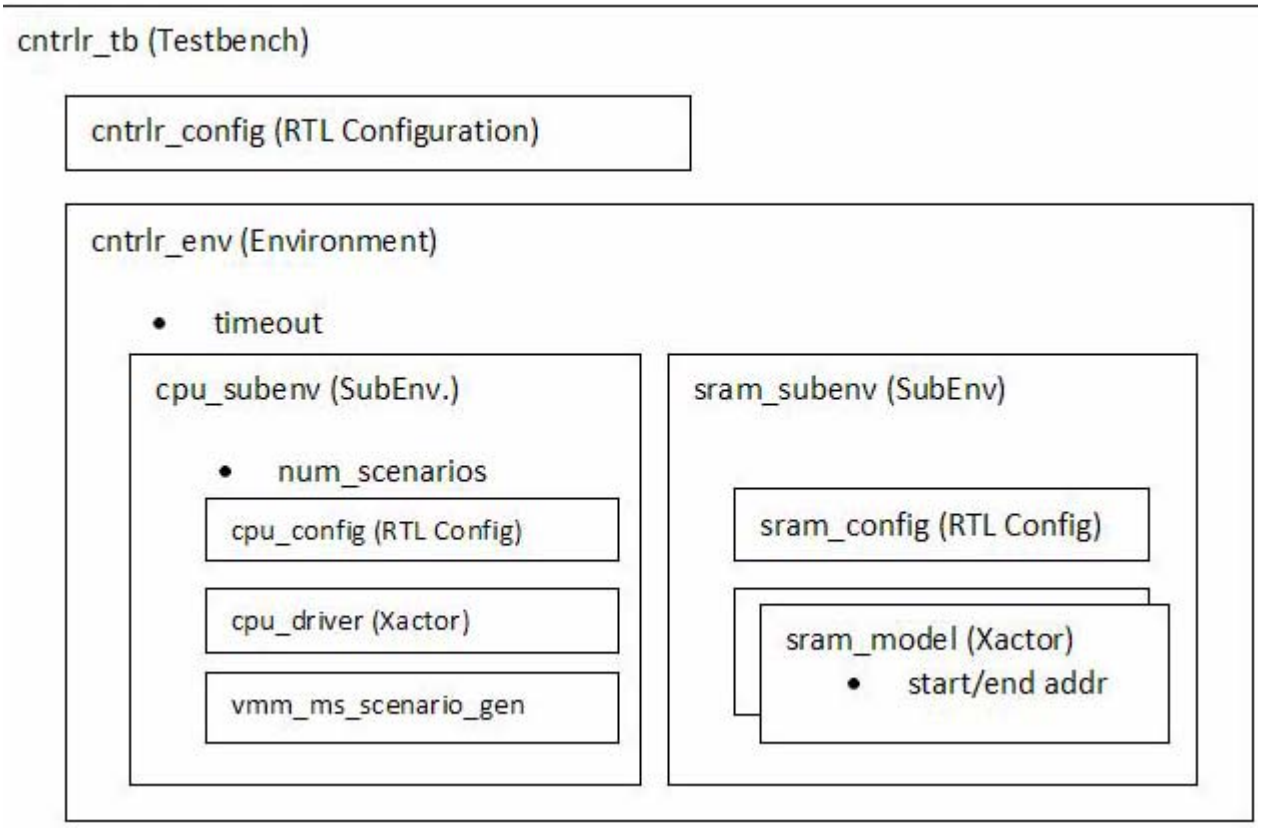
# Hierarchical Configuration Primer

## Introduction

Configurations are important elements of a verification environment. They are aimed at configuring various testbench components as well as the DUT. In addition to the usual ways of setting them with assignment statements and/or declarative constraints in objects, VMM allows configurations to be set from the simulator command line or a options file as well as from different hierarchies in the verification environment.

This primer describes how the `vmm_opts` utility class helps to pass values from the command line, option file during runtime and from the source code across hierarchies.

*Figure 10-2 Environment Hierarchy*




The testbench comprises of a few key elements, some of which are configuration-related. At the environment level, a time-out configuration parameter is defined, which defines how long the environment should be allowed to run if different verification components do not consent to the test to be completed for specific reasons.

The top-environment instantiates two subenvironments, the CPU subenvironment and the SRAM subenvironment. In the CPU subenvironment level, one of the configuration parameters is `num_scenarios`, which defines the number of scenarios to generate by the scenario generator.

The following sections walks you through different ways how global, hierarchical and structural options can be set and the values collected from within the code. Except for specifying instance paths through match patterns in the case of hierarchical options and structural options, the techniques of setting such options is the same across all three modes. Therefore, setting these different type of configuration options are discussed at the end.

## Step 1: Setting/Getting Global Options

VMM supports global options setting from the command line as well as from different code heirarchies and collecting them through by using `vmm_opts::get_*()` methods. As this is a global option, it has only one value which is more applicable for global environment variables such as, verbosity control, channel record/playback facility, etc.

Global options such as the simulation timeout limit generally set in the pre-test timeline retreiving these values. The `configure_ph()` phase is the recommended for capturing these values in the code.

You can change the global options in the command line, or in a command file, or by using the `set_*()` method in the testbench code. When global options are set from either the command line or from within the code, only the absolute value is specified and there is no need to specify the instance path hierarchy through the match patterns as global options are applicable throughout the complete environment.

Example 10-13 explains how to use global option for defining an environment parameter.

*Example 10-13 Getting Environment Timeout Option*

```
class cntrlr_env extends vmm_group;
   int timeout;
   function void configure_ph();
         timeout = vmm_opts::get_int("TIMEOUT",
                           100_000_000,
                              "Simulation Timeout Limit");
   endfunction
endclass
```

This provides the capability to specify these configuration options at run time using the syntax similar to `+vmm_opts+TIMEOUT=[value]` or `+vmm_opts_TIMEOUT=[value]`.

Another way to set the the value is through the `set_*()` methods in the code, however, for this specific example, you should set in a phase before the `configure_ph`.

## Step 2: Setting/Getting Hierarchical Options

In the previous step, you learnt how to assign a global option and then collect the value in the code. In some cases, you might want to specify an option on an instance basis or to specify the option either in code, command line or in a file. This step describes on how VMM can provide a set of `get_object_*()` methods that allows you to handle these situations.

You can use the `get_object_*()` methods to detect whether the named options are specified by the command line and assigns the parameters with the specified option values or default values. This provides the capability to specify these configuration options at run time using the syntax similar to `+vmm_opts+name=[value]` or `+vmm_opts_name=[value]`.

Most of the arguments are identical between the `get_object_*()` method and the corresponding `get_*()` method except that the `get_object_*()` methods have an output `is_set` argument which is set to TRUE if an explicit value is specified for the option. It also has an input `obj` argument which allows you to specify the object instance for the given option. The former is useful to test whether a specific option has been explicitly set to. The latter enables the hierarchical options.

The object hierarchy is specified using the custom regular expression defined in VMM. Given that most of these hierarchical options can be leveraged by the testcase to modify testbench behaviour across different instances, you should typically use the `vmm_opts::get_object_*` method after the `configure_test_ph()`, where the options would mostly be set by the testcases.

*Example 10-14 Getting Hierarchical Option*

```
class vip extends vmm_xactor;
bit b;
int i;
function start_of_sim_ph();
bit is_set;
b = vmm_opts::get_object_bit(is_set, this, "B",
"SET b value", 0);
i = vmm_opts::get_object_int(is_set, this, "I", 0,
"SET i value", 0);
endfunction
endclass
```

## Step 3: Getting Structural Options

In the previous step, you learnt how to use hierarchical options.

You will now learn how to assign options that affect the structure of the testbench components in the verification environment, a.k.a structural options. For instance, configuring transactors could be necessary for setting memory spaces, number of transactions or scenarios, protocol-specific parameters, etc.

Structural options should be declared in the transactors that extend `vmm_xactor` or the environments/subenvironments that extend `vmm_group`.

VMM provides a set of shorthand macros called `` `vmm_unit_config_*() `` to take advantage of the `vmm_opts::get_object_*()` methods. When the `` `vmm_unit_config_begin() `` and `` `vmm_unit_config_end `` macros are used, it is ensured that that these methods are called in the right phase, i.e. in `configure_ph()`. You should use these macros to declare these structural options and to provide default values. These structural options can then be set by using the `vmm_opts::set_*()` methods procedurally or with the run-time command line arguments or option files.

Example 10-15 shows how to model a cpu subenvironment with some structural options. The `enable_gen` option is non-random and specifies whether the generator is enabled. The `num_scenarios` option is randomized and specifies number of scenarios to be generated.

*Example 10-15 Getting Structural Options*

```
class cpu_subenv extends vmm_group;
   `vmm_typename (cpu_subenv)
   bit enable_gen;
   rand int num_scenarios;
   vmm_ms_scenario_gen gen;

   function void configure_ph();
```

```
    `vmm_unit_config_bit(enable_gen, 1,
              "Enable/disable the scenario generator",
                   0, DO_ALL);
  endfunction
  function void start_of_sim();
    `vmm_unit_config_rand_int (num_scenarios, 1,
                        "runs n scenarios", 0, DO_ALL);
    void'(this.randomize());
      this.gen.stop_after_n_scenarios = num_scenarios;

  endfunction
endclass
```

A shorthand `vmm_unit_config_bit macro is used for the enable_gen option. The first argument is the option itself. The second argument is the default value, if it is not set. The verbosity, the second last argument, is set to 0. DO_ALL in the last argument implements all the built-in methods such as copy() and allocate().

Similarly, a shorthand `vmm_unit_config_rand_int macro is used for the num_scenarios option. The `vmm_unit_config_rand* macro also sets the rand_mode of the variable to 0, so that the value set through configuration will not change due to randomization.

Using the shorthand macro to declare the num_scenarios option in the start_of_sim_ph() phase instead of the configure_ph() is deliberate. A lot of this has to do with how the testbench use model is defined. It is intended for each test in the testbench to have the ability to modify the number of random scenarios at the beginning of the test. To allow the tests to call set_int() method to modify this option at the beginning of the test, i.e. in the configure_test_ph() phase of the test timeline, this num_scenario option is assigned here using the shorthand

`vmm_unit_config_rand_int()` macro. This is important when you concatenate the multiple tests with different numbers of scenario.

## Step 4: Setting Options

There are three different ways to set options, in the order of precedence,

- Use the `set_*()` methods
- Use the option file
- Use the command line arguments

## Step 4a: Setting Options with `set_*`

You can use the `set_*()` methods to specify the options in a given hierarchy path pattern directly. If there is a pattern match to options, they are set using direct assignments. If there is no match, random options get randomized with the `randomize()` method and non-random options get the specified default values.

*Example 10-16 Setting Options in Code*

```
class test_random extends vmm_test;

  `vmm_typename (test_random);
   ...
   virtual function void configure_test__ph ();
     vmm_opts::set_int ("%*:num_scenarios", 50);
   endfunction

endclass
```

The num_scenarios option is matched with the%* hierarchical name pattern and is set to 50 for the test, test_random. For more details on the match patterns and how they can be used to specify select hierarchies, see “Simple Match Patterns” .

As the detection of setting the num_scenarios option is done in the start_of_sim_ph() phase, as long as the set_int() method is called before that, for example, the build_ph() phase in the pre-test timeline or configure_test_ph() phase in the test timeline, the specified option value will be in effect for the test.

## Step 4b: Setting Options in Command Line

The test to run and the num_scenarios options can be specified in the command line.

For example,

```
% simv +vmm_test=test_random \
       +vmm_num_scenarios=5

% simv +vmm_test=test_random \
       +vmm_opts+num_scenarios=5
```

The simulation will run for 5 scenarios, instead of 50, specified in the test. Overriding the options in the command line takes a high precedence than the set_*() methods.

## Step 4c: Setting Options With Command File

VMM options can be provided using the option files as well, where options are provided using +opt_name.

```
// file:    prj_opts.txt
```

```
+num_scenarios=10
```

Any combinations of the above methods can be used to set the options. For example, the command below runs the `test_random` test, uses the option file to configure the `num_scenarios` option, and uses the command line argument to set the simulation time-out to 99999.

```
% simv +vmm_test=test_random \
       +vmm_opts_file+prj_opts.txt \
       +vmm_opts+timeout=99999
```

The command below shows the precedence of the three option setting methods. The command configures the cpu subenvironment to run 10 scenarios, not 50 as specified in the `test_random` class, and not 10 as specified in the option file.

```
% simv +vmm_test=test_random \
       +vmm_opts_file+prj_opts.txt \
       +vmm_opts+num_scenarios=10
```

The command line options have the highest precedence, then the option file, and last the `set_*()` methods in the code.

## Conclusion

VMM provides the `vmm_opts` utility class to configure options in any level of the testbench hierarchy. You can set these options dynamically from the command line, option file, or from procedurally from the testbench code.

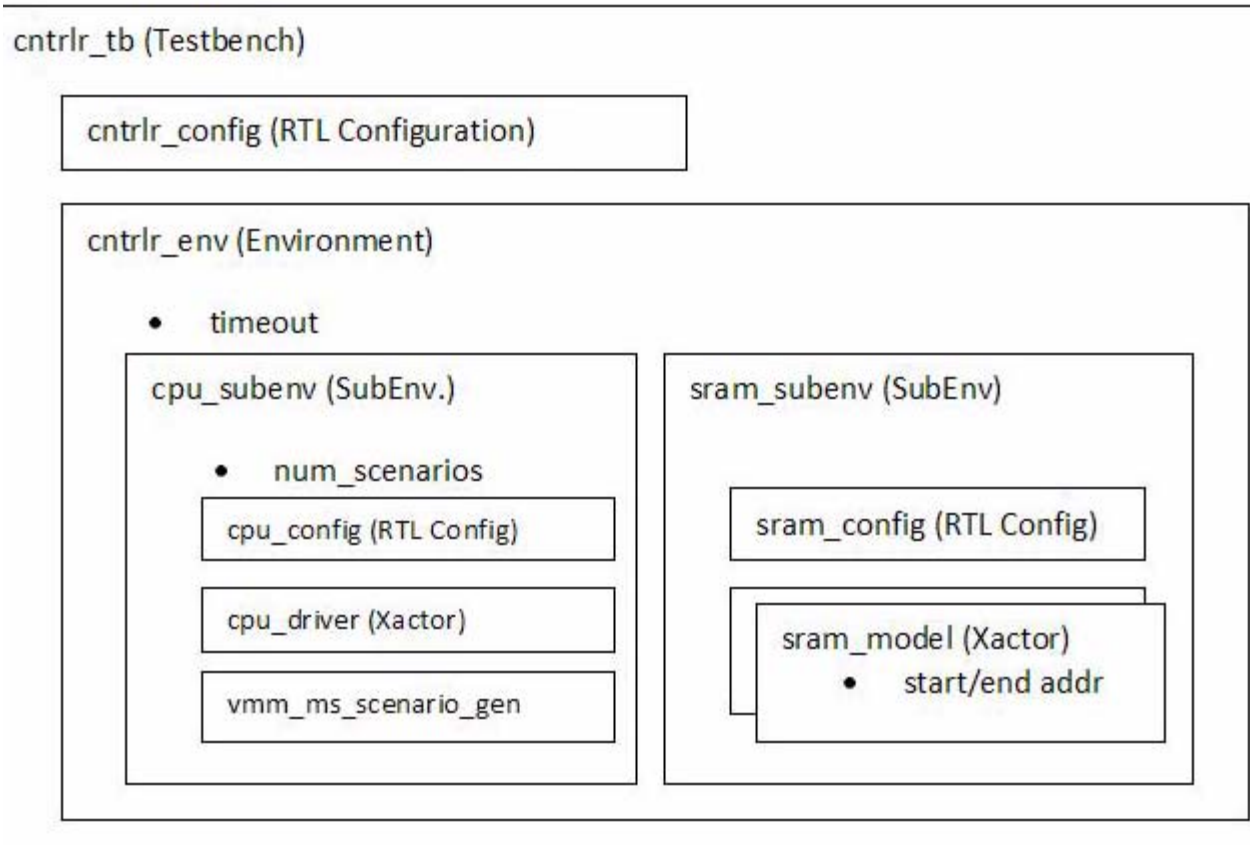
# RTL Configuration Primer

## Introduction

Unlike the configurations that affect the behavior of the testbench, there are some configuration parameters that define the RTL configuration. Examples of these configuration parameters are number of input ports, number of output ports, FIFO sizes and depths, etc. These configuration parameters must be shared with the testbench for consistent design verification.

RTL configuration depends upon an input file that describes the parameters for a given instance. While it is usual that these files are created manually, VCS can be used to randomize the RTL configurations and create one RTL configuration file for each randomized configurations. This feature helps verify the design in multiple configurations.

*Figure 10-3 Environment Hierarchy*




The testbench comprises several RTL configurations in different levels of hierarchy. The two subenvironments have their own RTL configurations: the CPU subenvironment has a CPU RTL configuration, `cpu_config`, and the SRAM subenvironment has a SRAM RTL configuration, `sram_config`. The top-level RTL configuration, `cntrlr_config`, simply instantiates the lower-level RTL configurations and makes them random.

## Step 1: Defining RTL Configurations

VMM provides the base class for RTL configuration called, `vmm_rtl_config`. You can use the extensions of `vmm_rtl_config` to encapsulate and define the RTL configuration parameters for the design.

The following RTL configuration parameters for this design are explained:

- CPU Address Width: This is the physical address width of the CPU interface on the DUT.
- Number of SRAM devices: This DUT can be configured to physically connect to a single, 2, or 4 SRAM devices, etc.

You can use the pre-defined macros, `` `vmm_rtl_config_* `` to specify the mapping between the configuration variable and the corresponding string in the configuration file.

Note: These macros should be defined between `` `vmm_rtl_config_begin `` and `` `vmm_rtl_config_end ``.

*Example 10-17 Implementing RTL Configuration*

```
class cpu_config extends vmm_rtl_config;
  rand int addr_width = 32;
  `vmm_rtl_config_begin (cpu_config)
    `vmm_rtl_config_int (addr_width, addr_width)
  `vmm_rtl_config_end (cpu_config)
  function new (string name = "",
                vmm_rtl_config parent = null);
    super.new (name, instance);
  endfunction
endclass

class sram_config extends vmm_rtl_config;
  rand int num_sram_devices;
```

```
  ...
  constraint cst_sram_config_valid {
    num_sram_devices inside {1, 2, 4};
  }
  `vmm_rtl_config_begin (sram_config)
      `vmm_rtl_config_int (num_sram_devices,
                           num_sram_devices)
  `vmm_rtl_config_end (sram_config)
   function new (string name = "",
                 vmm_rtl_config parent = null);
      super.new (name, instance);
   endfunction
endclass
```

## Step 2: Nested RTL Configurations

The top-level configuration, `cntrlr_config` simply instantiates the two subenvironment RTL configuration objects.

```
class cntrlr_config extends vmm_rtl_config;
  rand cpu_config cpu_cfg;
  rand sram_configsram_cfg;

 function new (string name, vmm_rtl_config parent=null);
    super.new(name);
    cpu_cfg = new ("cpu_cfg", this);
    sram_cfg = new ("sram_cfg", this);
  endfunction

endclass
```

## Step 3: Instantiating RTL Configurations

You use the following two-pass process for the RTL configuration:

1. Generate the RTL configuration for the RTL (and testbench) to use and save it in a file. This can be done using VCS but it does not have to be.

2. The parameters in the RTL configuration file are then used to compile both the design and testbench (which contains compile-time conditional code using `ifdef/`endif or parameterized values). Simulations are then run with this RTL configuration to verify the design in this configuration.

In Example 10-18, the CPU subenvironment has a RTL configuration and the SRAM subenvironment has another RTL configuration. It associates the previously declared cpu_config object within the CPU subenvironment in the build_ph() phase.

*Example 10-18 Instantiating RTL Configuration in Environment*

```
class cpu_subenv extends vmm_group;
  `vmm_typename (cpu_subenv)
  cpu_config cfg;
  function void build_ph();
     $cast(this.cfg,
           vmm_opts::get_object_obj (is_set,
                                     this, "cpu_cfg"));
  endfunction
endclass

class sram_subenv extends vmm_group;
  `vmm_typename (sram_subenv)
  sram_config cfg;
  sram_model    rams[];
  function void build_ph();
    $cast (this.cfg,
           vmm_opts::get_object_obj(is_set,
                                    this, "sram_cfg"));
    this.rams = new [cfg.num_sram_devices];
     for (int i = 0; i < cfg.num_sram_devices; i++) begin
       this.rams[i] = new ( .. );
     end
  endfunction
```

```
endclass
```

These RTL configurations are used in the `build_ph()` phase because it affects the structural content of the verification environment, such as the construction of right number of the SRAM models in the environment above.

## Step 4: Generating RTL Configuration File

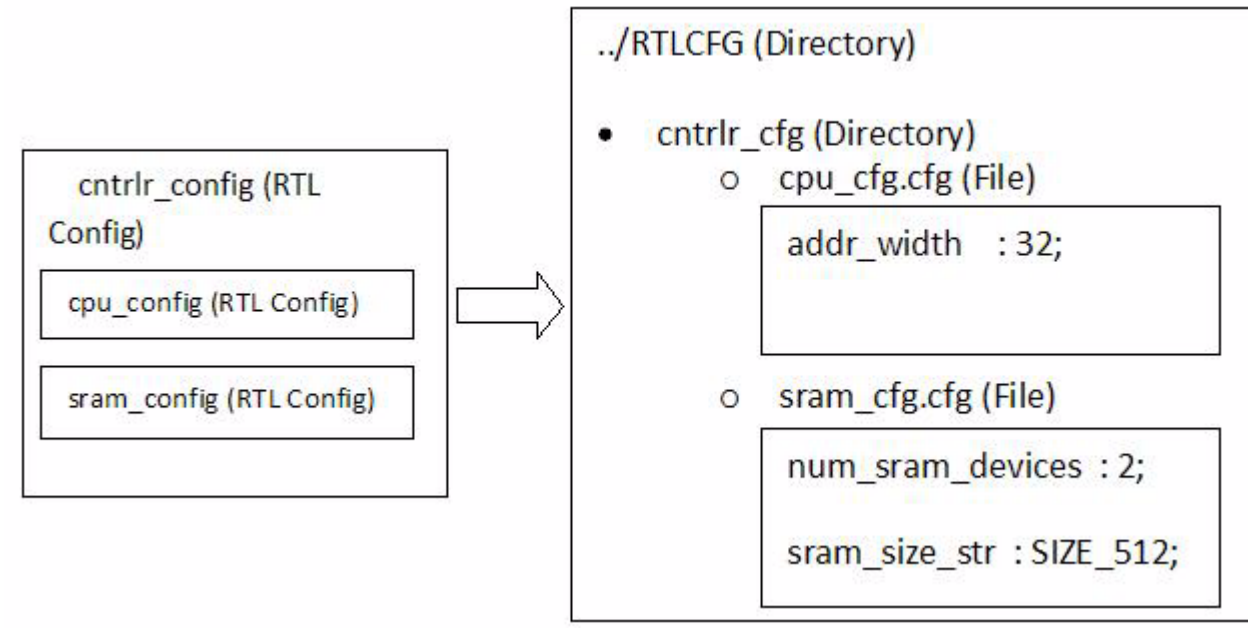
When you run the simulation with the `+vmm_gen_rtl_config` option, VCS considers all the objects that extend the `vmm_rtl_config` base class, create these configuration objects, randomize their contents, and write them out to multiple RTL configuration files, one for each `vmm_rtl_config` instance:

```
% simv +vmm_rtl_config=../RTLCFG +vmm_gen_rtl_config
```

At this point, no simulation is run during this pass. The RTL configuration files are written out in such a way that the hierarchical paths to the `vmm_rtl_config` instance are identical to the directory structure and RTL configuration files in it. The root directory for the RTL configuration files is specified by using the `+vmm_rtl_config` switch.

This creates a set of files in this example in a directory structure identical to the hierarchical structure of the RTL configuration objects in the testbench. An example of the file content is shown in Figure 10-4.

*Figure 10-4*




## Step 5: Simulation Using RTL Configuration File

VCS now needs to read a previously generated RTL configuration, which might contain multiple RTL configuration files from the directory specified by the `+vmm_rtl_config` switch.

```
% simv +vmm_rtl_config=../RTLCFG +vmm_test=test_random
```

Other VCS runtime arguments can be added to the command line to kick off one or more simulations with the same `"RTLCFG"` configuration.

## Conclusion

Some designs in RTL are configurable using `` `ifdef/`endif `` or parameter values, which must all be set before simulation runs. VMM provides the capability to randomize and generate the RTL configuration to use and then in a separate pass, verify the design in the specified configuration. The RTL configuration can also be created manually with directed parameters.

The RTL configuration files are organized in the directories and subdirectories representing the same object hierarchies in the testbench. You can customize the file format by using the file format class.

# Implicitly Phased Master Transactor Primer

## Introduction

This primer explains how to write VMM-compliant implicitly phased master transactor. A master transactor is a transaction-level interface on one side and pin wiggling on the other, also known as bus-functional models (BFM).

This section provides a step-by-step recommendations for implementing a command-layer transactor. As such, you should read it in a sequential fashion. You can use the same sequence to create your own specific transactor.

## The Protocol

The protocol used in this primer is the AMBA™ Peripheral Bus (APB) protocol. It is a simple single-master address-based parallel bus providing atomic individual read and write cycles. The protocol specification can be found in the AMBA™ Specification (Rev 2.0) available from ARM (http://arm.com).

When writing a reusable transactor, you should think of all possible applications it might be used in and not just the device you are using it for the first time. Therefore, even though the device in this primer only supports 8 address bits and 16 data bits, the APB transactors should be written for the entire 32-bit of address and data information.

## The Verification Components

Figure 10-5 illustrates the various components that are created throughout this primer. A command-layer master transactor interfaces directly to the DUT signals and initiates transactions upon requests on a transaction-level interface.

*Figure 10-5*

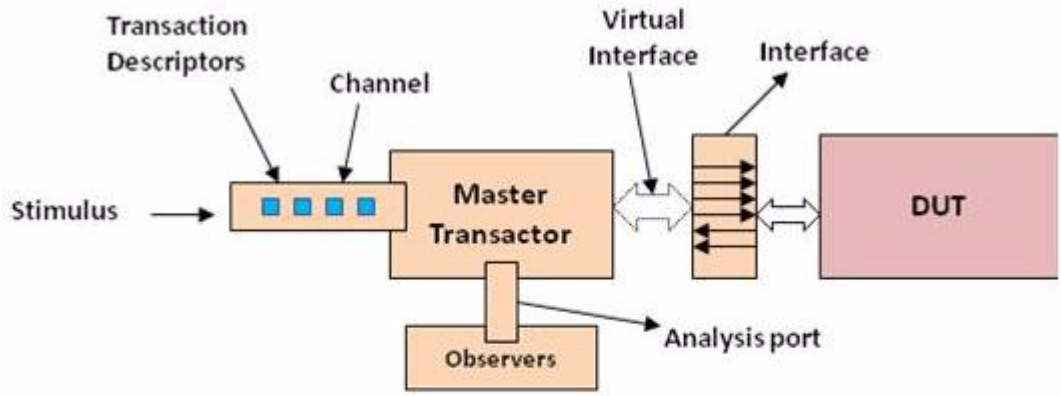



Figure 1

## Step 1: Implementing the APB Interface

The first step is to define the physical signals used by the protocol to exchange information between a master and a slave. A single exchange of information such as, READ or WRITE operation along with address or data is called a transaction. There might be multiple slaves on an APB bus but there can only be one master. Slaves are differentiated by responding to different address ranges.

The signals are declared inside an `interface`. The name of the `interface` is prefixed with `"apb_"` to identify that it belongs to the APB protocol. The entire content of the file declaring the interface is embedded in an `` `ifndef/`define/`endif `` construct. This allows the file to be included multiple times, whenever required, without causing multiple-definition errors.

The first step is to define the physical signals used by the protocol to exchange information between a master and a slave. A single exchange of information (READ or WRITE operation) is called a transaction. There may be multiple slaves on an APB bus but there can only be one master. Slaves are differentiated by responding to different address ranges.

The signals listed in the AMBA™ specification in section 2.4 are declared as wires inside the interface [Line 4-9].

Because this is a synchronous protocol, clocking blocks are used to define the direction and sampling of the signals [Line 10-13].

The clocking block defining the synchronous signals is specified in the modport for the APB master transactor. The clock signal need not be specified as it is implicit in the clocking block [Line 14].

The `interface` declaration is now sufficient for writing a master APB transactor. To be fully compliant, it should eventually include a modport for a slave and a passive monitor transactor. These can be added later when those transactors are written.

```
1  `ifndef APB_IF__SV
2  `define APB_IF__SV

3  interface apb_if(input bit pclk);
4    wire [31:0] paddr;
5    wire psel;
6    wire penable;
7    wire pwrite;
8    wire [31:0] prdata;
9    wire [31:0] pwdata;

10   clocking mck @(posedge pclk);
11     output paddr, psel, penable, pwrite, pwdata;
12     input prdata;
13   endclocking: mck

14   modport master(clocking mck);
15 endinterface: apb_if
16 `endif
```

To make the transactor component reusable across test benches, the physical level interface is modeled in two steps. This removes any dependency between the test, env and the DUT interface.

- Create an object wrapper for the virtual interface and make it as one of the properties of the transactor.
- Set this object using VMM configuration options either from the enclosing environment or from the top level.

```
class apb_master_port extends vmm_object;
   virtual apb_if.master mstr_if;
   function new(string name,
                virtual apb_if.master mstr_if);
```

```
      super.new(null, name);
      this.mstr_if = mstr_if;
   endfunction
endclass
```

## Step 2: Instantiating and Connecting the DUT

The interface can now be connected to the DUT. It is instantiated in a top-level module, alongside of the DUT instantiation. The connections to the DUT pins are specified using a hierarchical reference to the wires in the interface instance [Line 5-13].

This top-level module also contains the clock generators; using the bit type and ensuring that no clock edges occurs at time zero [Line 2, 3, 12 and 13].

```
// File: Command_Master_Xactor/tb_top.sv
1 module tb_top;
2   bit clk = 0;
3   apb_if apb0(clk);
    ...
4   my_design dut(...,
5     .apb_addr   (apb0.paddr),
6     .apb_sel    (apb0.psel),
7     .apb_enable (apb0.penable),
8     .apb_write  (apb0.pwrite),
9     .apb_rdata  (apb0.prdata),
10    .apb_wdata  (apb0.pwdata),
      ...
12    .clk       (clk));

13  always #10 clk = ~clk;

14  endmodule: tb_top
```

## Step 3: Modeling the APB Transaction

The next step is to define the APB transaction descriptor. Traditionally, tasks would have been defined, one for the READ transaction and one for the WRITE transaction.

```
task read(input  bit   [31:0] addr,
          output logic [31:0] data);

task write(input bit [31:0] addr,
           input bit [31:0] data);
```

This works well for directed tests, but not at all for random tests. A random test requires a transaction descriptor. This descriptor is a class extended from the `vmm_data` class, containing a public `rand` property enumerating the directed tasks and public `rand` properties for each task argument. If an argument is the same across multiple tasks, a single property can be used. It also needs a static `vmm_log` property instance used to issue messages from the transaction descriptor. This instance of the message service interface is passed to the `vmm_data` constructor [Line 6-8] and it is done by using the shorthand macros.

Note how the same property is used for "data". It is interpreted differently depending on the transaction kind. In a WRITE transaction, it is interpreted as the data to be written. In a READ transaction, the random content is initially ignored and it is replaced by the data value that was read. The type for the "data" property is logic as it is a superset of the bit type and allows the description of READ cycles to reflect the unknown results.

A transaction-level interface is required to transfer transaction descriptors to a transactor to be executed. This is done by using the `` `vmm_channel `` macro [Line 16].

The transaction descriptor class can be factory enabled by using `vmm_class_factory macro. Factory Service provides an easy way to replace any kind of transaction object by a similar object. This replacement can take place from anywhere in the verification environment or in the test case [Line 14]. For details on factory, see “Step 1: Modeling Classes to be Factory Ready” .

The transaction descriptor class requires many utility methods implemented to facilitate various types of operation. All the necessary utility classes like `new, copy, compare, allocate, byte pack` and `unpack` etc. along with `vmm_log` instance can be automatically done using the standard shorthand macros. The shorthand macros implements the code for these methods.

The following code is the complete transaction descriptor class using the shorthand macros [Line 9-13]:

```
1   `ifndef APB_TRANS__SV
2   `define APB_TRANS__SV

3   `include "vmm.sv"

4   class apb_trans extends vmm_data;
5     `vmm_typename(apb_trans)

6     rand enum {READ, WRITE} kind;
7     rand bit   [31:0] addr;
8     rand logic [31:0] data;

9    `vmm_data_member_begin(apb_trans)
10    `vmm_data_member_scalar(addr, DO_ALL)
11    `vmm_data_member_scalar(data, DO_ALL)
12    `vmm_data_member_enum(kind, DO_ALL)
13    `vmm_data_member_end(apb_trans)

14    `vmm_class_factory(apb_trans)
15  endclass: apb_trans
```

```
16 `vmm_channel(apb_trans)
17 ...
18 `endif
```

## Step 4: Modeling the Master Transactor

In VMM implicit phasing, the environment steps are called as phases that are used to coordinate the simulation execution. Phases can be thought as testbench component activities which are activated and flow controlled through a timeline. VMM predefines several simulation phases. Transactors progress through a series of phases throughout the simulation. The phases of all transactors are automatically coordinated and executed synchronously with other transactors during simulation execution.

Implicit phasing works only with transactors that are based on the `vmm_group` or `vmm_xactor` base class. Also, note that the `stop_xactor` method is not called automatically and should be called in the `shutdown_ph` of this transactor. The enclosing environment is responsible for controlling the shutdown behavior of the transactor. You can override any of these phase methods to implement the required functionality for a particular testbench component.

For our APB master, the important phases are,

- `build_ph`
- `connect_ph`
- `start_of_sim_ph`
- `run_ph`
- `shutdown_ph`

You should use the main method to implement the transactor functionality and fork off the run phase to advance to the next phase.

The master transactor can now be started. It is a class derived from the `vmm_xactor` base class.

The transactor needs a transaction-level interface to receive transactions to be executed and a physical-level interface to wiggle pins. The former is done using `vmm_channel` instance and the latter is done by using a virtual modport wrapper class.

READ and WRITE tasks are implemented in this class. They are declared virtual so that the transactor might be extended to modify the behavior of these tasks if required. They are also declared protected to prevent them from being called from outside the class and create concurrent bus access problems.

```
class apb_master extends vmm_xactor;
  `vmm_typename(apb_master)
  apb_trans_channel     in_chan;
  apb_master_port       mstr_port_obj;
  virtual apb_if.master mstr_if;

  extern function new(string inst="",
                      vmm_unit parent=null);
  // Supporting tasks
  extern virtual protected task read(
       input bit [31:0] addr, output logic [31:0] data);
  extern virtual protected task write(
      input bit [31:0] addr, input bit [31:0] data);

  // Component phases
  extern virtual function void reset_xactor(
       reset_e rst_typ = SOFT_RST);
  extern virtual function void build_ph();
  extern virtual function void connect_ph();
  extern virtual function void start_of_sim_ph();
  extern virtual task run_ph();
```

```
  extern virtual task shutdown_ph();

  // Factory enablement
  extern virtual function apb_master allocate();
  extern virtual function apb_master copy();
  `vmm_class_factory(apb_master)
endclass: apb_master
```

It is important that this transactor gets associated with its parent. The main reason is to allow it to be replaced, queried or its options to be modified. This association is established in the constructor:

```
function apb_master::new(string inst="",
                         vmm_unit parent=null);
    super.new(get_typename(), inst, 0, parent);
endfunction
```

The `get_typename()` method returns the string `apb_master` . It is declared by the `` `vmm_typename(apb_master) `` macro.

When the transactor is reset, the input channel must be flushed and the critical output signals must be driven to their idle state. This is accomplished in the extension of the `vmm_xactor::reset_xactor()` method. This method might be called in the transactor constructor to initialize the output signals to their idle state, or explicit signal assignments might be used in the constructor.

```
virtual function void apb_master::reset_xactor(
                          reset_e rst_typ = SOFT_RST);
    super.reset(rst_typ);
    this.in_chan.flush();
    this.sigs.mck.psel    <= '0;
    this.sigs.mck.penable <= '0;
  endfunction: reset_xactor
```

An `analysis_port` is used to convey the transaction processed by this transactor to the other testbench components. In the `build` phase, construct the analysis port and TLM interfaces to associate this analysis port with this transactor so that you can back trace to it if necessary. You might need the TLM interfaces for passing transactions in a blocking/non-blocking way.

```
class apb_master extends vmm_xactor;
  `vmm_typename(apb_master)
  ...
  vmm_tlm_analysis_port#(apb_master, apb_trans)
      analysis_port;
  ...
  function void apb_master::build_ph();
    analysis_port = new(this,
               {get_object_name(), "_analysis_port"});
  endfunction: build_ph
endclass: apb_master
```

In the connect phase, the testbench components are connected to assist the data flow between the generation through the DUT pin interface.

In the APB transactor's `connect` phase, a handle to the virtual interface port is obtained using the `vmm_opts_get` method. Here, the virtual interface connection is made between the master transactor and the DUT pin interface.

```
virtual function void apb_master::connect_ph();
  bit is_set;
  if ($cast(this.mstr_port_obj,
            vmm_opts::get_object_obj(is_set,
                                     this,
                                       "apb_mstr_port")))
    begin
      if (mstr_port_obj != null)
        this.mstr_if = mstr_port_obj.mstr_if;
      else
       `vmm_fatal(log, "Virtual port wrapper not
```

```
initialized");
  end
endfunction: connect_ph
```

Implement the `start_of_sim` phase to ensure that the channel and the interface are present. If null, report a FATAL error message and exit the simulation.

```
function void apb_master::start_of_sim_ph();
  if (mstr_port_obj == null)
     `vmm_fatal(log, "Virtual port is not connected to
the actual interface instance");
endfunction
```

The transaction descriptors are pulled from the input channel and translated into method calls in the `main()` task. The most flexible transaction execution mechanism uses the active slot as it supports, block, non-blocking and out-of-order execution models.

Because the protocol supports being suspended between transactions, the `vmm_xactor::wait_if_stopped_or_empty()` method is used to suspend the execution of the transactor if it is stopped.

The main `apb` master transactor functionality is implemented in the `main()` task of the transactor. You have a choice to use the main method or the run phase to implement this behavior.

There is no difference between using the main method (VMM 1.1 style) over the run phase (VMM 1.2 style).

Note: After processing the current transaction, the analysis port's write method is called to issue the observed transaction to the other testbench components for further analysis.

```
task apb_master::run_ph();
  apb_trans tr;
  bit drop;
  fork
    forever begin
      this.wait_if_stopped_or_empty(this.in_chan);
      this.in_chan.activate(tr);
      ...
      this.in_chan.start();
      case (tr.kind)
        apb_trans::READ: this.read(tr.addr, tr.data);
         apb_trans::WRITE: this.write(tr.addr, tr.data);
      endcase
      this.in_chan.complete();
      ...
      this.analysis_port.write(tr);
      this.in_chan.remove();
    end
  join_none
endtask: run_ph
```

In the `shutdown_ph` phase, the `stop_xactor()` method is called.

Note: The enclosing testbench environment is responsible for controlling the shutdown phase of this transactor.

```
task apb_master::shutdown_ph();
  this.stop_xactor();
endtask
```

The READ and WRITE tasks are coded exactly as they would be if good old Verilog was used. It is a simple matter of assigning output signals to the proper value then sampling input signals at the right point in time. The only difference is that the physical signals are accessed through the clocking block of the virtual modport instead of pins on a module and they can only be assigned using non-blocking assignments.

Similarly, the active clock edge is defined by waiting on the clocking block itself, not an edge of an input signal in order to increase flexibility and reusability of the code.

```
protected task apb_master::read(input bit [31:0] addr,
                                output logic [31:0] data);
    this.sigs.mck.paddr  <= addr;
    this.sigs.mck.pwrite <= '0;
    this.sigs.mck.psel   <= '1;
    @ (this.sigs.mck);
    this.sigs.mck.penable <= '1;
    @ (this.sigs.mck);
    data = this.sigs.mck.prdata;
    this.sigs.mck.psel    <= '0;
    this.sigs.mck.penable <= '0;
endtask: read

protected task apb_master::write(input bit [31:0] addr,
                                 input bit [31:0] data);
    this.sigs.mck.paddr  <= addr;
    this.sigs.mck.pwdata <= data;
    this.sigs.mck.pwrite <= '1;
    this.sigs.mck.psel   <= '1;
    @ (this.sigs.mck);
    this.sigs.mck.penable <= '1;
    @ (this.sigs.mck);
    this.sigs.mck.psel    <= '0;
    this.sigs.mck.penable <= '0;

  endtask: write
```

To make the transactor factory enabled, you should use`vmm_class_factory macro.

Note: You must implement the `copy` and `allocate` method whenever using the factory for the transactor implementation.

```
function apb_master apb_master::copy();
  apb_master drv;
```

```
    drv = this.allocate();
    return drv;
  endfunction

  function apb_master apb_master::allocate();
    vmm_unit prnt;
    apb_master drv;
    $cast(prnt, this.get_parent_object());
    drv = new(this.get_object_name(), prnt);
    return drv;
  endfunction
```

The transactor as presently coded provides basic functionality. Now you have a transactor that can perform READ and WRITE cycles with identical capabilities to one you would have written using the old Verilog language.

The problem is that the transactor as coded is not very reusable. It is not be possible to modify the behavior of this transactor, for example, to introduce delays between transactions, to synchronize the start of a transaction with some other external event, or modify a transaction to inject errors-without modifying the transactor itself or constantly rewrite the `apb_master::read()` and `apb_master::write()` virtual methods.

A callback method allows you to extend the behavior of a transactor without having to modify the transactor itself. Callback methods should be provided before and after a transaction executes.

The "pre-transaction" callback method allows errors to be injected and delays to be inserted. The "post-transaction" callback method allows delays to be inserted and the result of the transaction to be recorded in a functional coverage model or checked against an expected response.

The callback methods are first defined as virtual tasks or virtual void functions in a callback façade class extended from the `vmm_xactor_callbacks` base class. It is a good idea to create a mechanism in the "pre-transaction" callback method to allow an entire transaction to be skipped or dropped.

```
typedef class apb_master;
class apb_master_cbs extends vmm_xactor_callbacks;
   virtual task pre_cycle(apb_master xactor,
                          apb_trans     cycle,
                          ref bit    drop);
   endtask: pre_cycle
   virtual task post_cycle(apb_master xactor,
                           apb_trans     cycle);
   endtask: post_cycle
endclass: apb_master_cbs
```

Next, the appropriate callback method needs to be invoked at the appropriate point in the execution of the transaction by using the `` `vmm_callback() `` macro. This is done in the `run_ph`.

```
task apb_master::run_ph();
  bit drop;
  fork
    forever begin
      drop = 0;
      `vmm_callback(apb_master_cbs,
                    pre_cycle(this, tr, drop));
      if (drop) begin
        `vmm_debug(log,
                   {"Dropping transaction...\n",
                    tr.psdisplay("")});
        this.in_chan.remove();
        continue;
      end
     `vmm_callback(apb_master_cbs,
                   post_cycle(this, tr));
    end
  join_none
endtask: run_ph
```

For complete code, see Appendix.

## Step 5: Implementing an Observer

The following example shows how to implement a simplified observer model that prints the observed transaction. This is instantiated in the testbench environment and the appropriate connections are made during the connect phase.

```
class observer extends vmm_object;
  `vmm_typename(observer)
  vmm_tlm_analysis_export#(observer, apb_trans) obsrv =
new(this,
                             "apb_trans_obsrv");

 function new(string inst="",vmm_object parent = null);
    super.new(parent, get_typename());
  endfunction
endclass: observer

 virtual function void write (int id = -1, apb_trans tr);
    tr.display("Trans Rcvd        ");
  endfunction: write
```

## Step 6: Instantiating the Components in the Environment

You use the `vmm_group` base class to implement the implicitly-phased `apb` environment. An instance of the `apb` master transactor, observer and an `APB` channel are instantiated here. The components are built in the build phase and connected to each other in the connect phase of the environment.

```
// File: apb/tb_env.sv
`ifndef TB_ENV__SV
`define TB_ENV__SV
```

```
`include "vmm.sv"
`include "apb.sv"

class tb_env extends vmm_group;
  `vmm_typename(tb_env)
  apb_master        mstr;
  apb_trans_channel gen_to_drv_chan;
  observer          obsrv_apb_trans;
  vmm_log log = new("log", "TB_ENV_LOG");

  function new(string inst, vmm_unit parent);
    super.new(get_typename(), inst, parent);
  endfunction

  extern function void build_ph();
  extern function void connect_ph();
endclass: tb_env

`endif
```

The component construction happens in the build phase. Environment components are allocated as required by the test bench. Factory service acts as a replacement for object construction. Rather than declaring an object and constructing it using its `new()` method, VMM provides facilities to construct and replace the objects as needed by the testbench.

Create object instance by using a method `class::create_instance()`, this object instance in turn becomes a factory, an object that can be replaced.

Replace this factory object by using another set of static methods for either copying or allocating a new object using, `class::override_with_copy()` or `class::override_with_new()` methods.

```
function void tb_env::build_ph();
  this.mstr = apb_master::create_instance(this,
                                          "APB_MSTR");
```

```
    this.gen_to_drv_chan = new("Gen2DrvChan", "apb_chan");
    this.obsrv_apb_trans = new("TRANS_OBSVR", this);
  endfunction: build_ph
```

The input channel of this transactor is connected in the connect phase of the environment as needed. TLM interface connection happens in this phase to bind the observer to the appropriate transactor.

```
function void tb_env::connect_ph();
  this.mstr.in_chan = this.gen_to_drv_chan;
  this.mstr.analysis_port.tlm_bind(
                                obsrv_apb_trans.obsrv);
endfunction: connect_ph
```

## Step 7: Implementing Sanity Test

You use the VMM configuration options to set and get the virtual interface port objects. The environment is created and the interface port object is set in the initial block of the main program.

```
// File: apb/apb_tb.sv
`include "test_simple.sv"
program automatic apb_tb;
  tb_env env ;
  apb_master_port apb_mstr_p0;
  initial begin
    env = new("TB_ENV");
    apb_mstr_p0 = new("apb_port", tb_top.apb_if_p0);
    // Set the master port interface
    vmm_opts::set_object("APB_MSTR:apb_mstr_port",
                         apb_mstr_p0, env);
  end
endprogram
```

A simple test to perform a write followed by a read of the same address can now be written and executed to verify the correct operation of the transactor and the DUT interface. The test is written by extending the `vmm_test` base class.

The directed stimulus is created by instantiating transaction descriptors appropriately filled. It is a good idea to randomize these descriptors by only constraining those properties that are needed for the directed test.

By doing this, any additional property is randomized instead of defaulting to always the same value.

```
// File: apb/test_simple.sv
`include "tb_env.sv"
class apb_test1 extends vmm_test();
  `vmm_typename(apb_test1);
  tb_env env;
  function new (string name = "APB_TEST1", tb_env env);
    super.new(name);
    this.env = env;
  endfunction

  task run_ph();
    apb_trans rd, wr;
    bit ok;

    // Transaction : Write
    wr = new;
    ok = wr.randomize() with {kind == WRITE;};
    if (!ok)
      `vmm_fatal(log, "Unable to randomize WRITE cycle");
    env.mstr.in_chan.put(wr);

    // Transaction : Read
    rd = new;
    ok = rd.randomize() with {
       kind == READ;
       addr == wr.addr;
```

```
      };
      if (!ok)
        `vmm_fatal(log, "Unable to randomize READ cycle");
      env.mstr.in_chan.put(rd);

      // Compare the Read data
      if (rd.data[15:0] !== wr.data[15:0]) begin
        `vmm_error(log, "Readback value != write value");
      end
    endtask: run_ph
  endclass
```

You can run this test many times, each time with a different seed to verify the transactor and the DUT using different addresses.

To simplify, the test scenario is added directly in the run phase of example. However, it is recommended to use MSSG for achieving the same result.

## Step 8: Adding Debug Messages

To be truly reusable, it should be possible to understand what the transactor does and debug its operation without having to inspect the source code. This capability might even be a basic requirement if you plan on shipping encrypted or compiled code.

Debug messages should be added at judicious points to indicate what the transactor is about to do, is doing or has done. These debug messages are inserted using the `` `vmm_trace() ``, `` `vmm_debug() `` or `` `vmm_verbose() `` macros.

To use these debug messages, see master transactor code in Appendix.

## Step 9: Implementing Transaction Generator

To promote the use of random stimulus, it is a good idea to pre-define random transaction generators whenever transaction descriptors are defined.

It is simple to use the `` `vmm_atomic_gen() `` and `` `vmm_scenario_gen() `` macros in the transaction descriptor file. These macros automatically create generators that follows all the guidelines for any user-defined type.

The Multi Stream Scenario Generator (MSSG) base class component provides the capability to implement hierarchical and reusable transaction scenarios. It controls and coordinates existing scenarios to achieve a fine-grained control over stimulus.

For details, see Chapter 6, "Implementing Tests & Scenarios".

## Step 10: Implementing the Top-Level File

To include all the necessary files without having to know the detailed filenames and file structure of the transactor, interface and transaction descriptor, it is a good idea to create a top-level file that automatically includes all the source files that make up the verification IP for a protocol.

```
// File: apb/apb_tb_files.sv
`ifndef APB__SV
`define APB__SV
`include "vmm.sv"
`include "apb_if.sv"
`include "apb_rw.sv"
`include "apb_master.sv"
`endif
```

In this example, only a master transactor is implemented; but a complete VIP for a protocol would also include a slave transactor and a passive monitor transactor. All of these transactors would be included in the top-level file.

## Step 11: Congratulations!

You have now completed the creation of VMM-compliant command-layer master transactor.

Upon reading this primer, you probably realized that there is much code that is similar across different master transactors. Wouldn't be nice if you could simply cut-and-paste from an existing VMM-compliant master transactor and only modify what is unique or different for your protocol? That can easily be done using the "vmmgen" tool provided with VCS. Based on a few simple question and answers, it will create a template for various components of a VMM-compliant master transactor.

You may consider reading other publications in this series to learn how to write VMM compliant command-layer slave transactors, command-layer passive monitor transactors, functional-layer transactors or verification environments.

## Appendix A

The following example is the complete code of the APB master.

File: apb/apb_master.sv

```
`ifndef APB_MASTER__SV
`define APB_MASTER__SV

`include "apb_if.sv"
`include "apb_trans.sv"

////// Transactor Extension Callback methods /////
typedef class apb_master;
class apb_master_cbs extends vmm_xactor_callbacks;
   virtual task pre_cycle(apb_master xactor,
                          apb_trans     cycle,
                          ref bit    drop);
   endtask: pre_cycle

   virtual task post_cycle(apb_master xactor,
                           apb_trans     cycle);
   endtask: post_cycle
endclass: apb_master_cbs

////////////// APB Master Driver Class //////////
class apb_master extends vmm_xactor;
  `vmm_typename(apb_master)

  // Variables declaration
  apb_trans_channel  in_chan;
  apb_master_port mstr_port_obj;
  virtual apb_if.master mstr_if;

  // Analysis port
  vmm_tlm_analysis_port#(apb_master, apb_trans)
analysis_port;

  // Component phases
  extern function new(string inst="", vmm_unit
```

```
parent=null);
  extern virtual function void reset_xactor(reset_e
rst_typ = SOFT_RST);
  extern virtual function void build_ph();
  extern virtual function void connect_ph();
  extern virtual function void start_of_sim_ph();
  extern virtual task run_ph();
  extern virtual task shutdown_ph();

  // Supporting tasks
  extern virtual protected task read(input  bit   [31:0]
addr, output logic [31:0] data);
  extern virtual protected task write(input bit [31:0]
addr, input bit [31:0] data);

  // Factory enablement
  extern virtual function apb_master allocate();
  extern virtual function apb_master copy();
  `vmm_class_factory(apb_master)

endclass: apb_master

////////////// Constructor /////////////////
function apb_master::new(string inst, vmm_unit parent);
  super.new(get_typename(), inst, 0, parent);
endfunction

//////////////// Build Phase /////////////////
function void apb_master::build_ph();
  // Construct the analysis port
  analysis_port = new(this, {get_object_name(),
"_analysis_port"});
endfunction: build_ph

////////////// Connect Phase ////////////////
function void apb_master::connect_ph();
  begin
    bit is_set;
      `vmm_note(log,$psprintf("**** %s: Entering
connect_ph ....\n",get_object_hiername()));
      //
    if ($cast(this.mstr_port_obj,
```

```
vmm_opts::get_object_obj(is_set,this,"apb_mstr_port")))
begin
       if (mstr_port_obj != null)
        this.mstr_if = mstr_port_obj.mstr_if;
       else
        `vmm_fatal(log, "Virtual port [Master] wrapper
not initialized");
    end
      // Initialize the port signals
    this.mstr_if.mck.psel    <= '0;
    this.mstr_if.mck.penable <= '0;
      //
      `vmm_note(log,$psprintf("**** %s: Exiting
connect_ph ....\n",get_object_hiername()));
  end
endfunction: connect_ph

////////////// start_of_sim_ph Phase ////////
function void apb_master::start_of_sim_ph();
   if (mstr_port_obj == null)
      `vmm_fatal(log, "Virtual port is not connected to
the actual interface instance");
endfunction
//
function void apb_master::reset_xactor(reset_e rst_typ =
SOFT_RST);
  super.reset_xactor(rst_typ);
  this.in_chan.flush();
  this.mstr_if.mck.psel    <= '0;
  this.mstr_if.mck.penable <= '0;
endfunction: reset_xactor
//
task apb_master::run_ph();
  apb_trans tr;
  bit drop;
  fork
    forever begin
      //
      this.wait_if_stopped_or_empty(this.in_chan);
      this.in_chan.activate(tr);
      @ (this.mstr_if.mck);
      drop = 0;
```

```
      `vmm_callback(apb_master_cbs, pre_cycle(this, tr,
drop));
      if (drop) begin
        `vmm_debug(log, {"Dropping transaction...\n",
tr.psdisplay(" ")});
        this.in_chan.remove();
        continue;
      end
      `vmm_trace(log, {"Starting transaction...\n",
tr.psdisplay(" ")});
       this.in_chan.start();
       case (tr.kind)
         apb_trans::READ : this.read(tr.addr, tr.data);
        apb_trans::WRITE: this.write(tr.addr, tr.data);
       endcase
       this.in_chan.complete();
       `vmm_trace(log, {"Completed transaction...\n",
tr.psdisplay(" ")});
       `vmm_callback(apb_master_cbs, post_cycle(this,
tr));
       // broadcast to the observers
       this.analysis_port.write(tr);
       this.in_chan.remove();
    end
  join_none
endtask: run_ph

////////////// shutdown_ph Phase ///////////
task apb_master::shutdown_ph();
   this.stop_xactor();
endtask

////////////// Factory: copy ///////////
function apb_master apb_master::copy();
   apb_master drv;
   drv = this.allocate();
   return drv;
endfunction

////////////// Factory: allocate /////////
function apb_master apb_master::allocate();
   vmm_unit prnt;
```

```
   apb_master drv;
   $cast(prnt, this.get_parent_object());
   drv = new(this.get_object_name(), prnt);
   return drv;
endfunction

////////////// APB BUS READ task /////////
task apb_master::read(input  bit   [31:0] addr, output
logic [31:0] data);
  this.mstr_if.mck.paddr   <= addr;
  this.mstr_if.mck.pwrite  <= '0;
  this.mstr_if.mck.psel    <= '1;
  @ (this.mstr_if.mck);
  this.mstr_if.mck.penable <= '1;
  @ (this.mstr_if.mck);
  data = this.mstr_if.mck.prdata;
  this.mstr_if.mck.psel    <= '0;
  this.mstr_if.mck.penable <= '0;
endtask: read

////////////// APB BUS WRITE task ////////
task apb_master::write(input bit [31:0] addr, input bit
[31:0] data);
  this.mstr_if.mck.paddr   <= addr;
  this.mstr_if.mck.pwdata  <= data;
  this.mstr_if.mck.pwrite  <= '1;
  this.mstr_if.mck.psel    <= '1;
  @ (this.mstr_if.mck);
  this.mstr_if.mck.penable <= '1;
  @ (this.mstr_if.mck);
  this.mstr_if.mck.psel    <= '0;
  this.mstr_if.mck.penable <= '0;
endtask: write
//
`endif
```

# A

# Standard Library Classes (Part 1)

This appendix provides detailed information about the OpenVera and SystemVerilog classes that compose the VMM Standard Library. The functionality of OpenVera and SystemVerilog classes is identical, except for the following difference:

- OpenVera methods have a prefix of `rvm`
- SystemVerilog methods have a prefix of `vmm`

Note:

Each method, explained in this appendix, uses the SystemVerilog name in the heading to introduce it. Additionally, there are a few instances where a `_t` suffix is appended to indicate that it may be a blocking method.

Usage examples are specified in a single language, but that should not prevent the use of the other language, as both the languages are almost identical. Rather than providing usage examples that are almost identical, this appendix provides different examples for each language.

The classes are documented in alphabetical order. The methods in each class are documented in a logical order, where methods that accomplish similar results are documented sequentially. A summary of all available methods, with cross-references to the page where their detailed documentation can be found, is provided at the beginning of each class specification.

## VMM Standard Library Class List


- "factory"
- "vmm_atomic_gen#(T)"
- "<class-name>_atomic_gen_callbacks"
- "vmm_atomic_scenario#(T)"
- "vmm_broadcast"
- "vmm_channel"
- "vmm_connect#(T,N,D)"
- "vmm_consensus"
- "vmm_data"
- "vmm_env"
- "vmm_group"

- “vmm_group_callbacks”
- “vmm_log”
- “vmm_log_msg”
- “vmm_log_callback”
- “vmm_log_catcher”
- “vmm_log_format”
- “vmm_ms_scenario”
- “vmm_ms_scenario_gen”
- “vmm_notification”
- “vmm_notify”
- “vmm_notify_callbacks”
- “vmm_notify_observer#(T,D)”
- “vmm_object”
- “vmm_object_iter”
- “vmm_opts”

# factory

The `factory` class is the utility class to generate instances of any class through the factory mechanism.

## Summary


- factory::create_instance() ..... page A-5
- factory::override_with_new() ..... page A-7
- factory::override_with_copy() ..... page A-9
- factory::this_type() ..... page A-11
- `vmm_class_factory(classname) ..... page A-12

## factory::create_instance()

Creates an instance of the specified class type.

### SystemVerilog

```
static function classname
classname::create_instance(vmm_object parent, string name,
   string fname = "", int lineno = 0);
```

### Description

Creates an instance of the specified class type, for the specified name in the scope, created by the specified parent `vmm_object`.

The new instance is created by calling `allocate()` or `copy()` on the corresponding `factory` instance, specified using the `override_with_new()` or `override_with_copy()` method, in this class, or any of its parent (base) classes. If you do not specify any `factory` instance, then it creates an instance of this class.

The newly created instance contains the specified name and the specified `vmm_object` as parent, if the newly created instance is extended from `vmm_object`.

The `fname` and `lineno` arguments are used to track the file name and the line number where the instance is created using `create_instance`.

### Example

```
class ahb_trans extends vmm_object;
   `vmm_class_factory(ahb_trans)
endclass
class ahb_gen extends vmm_group;
```

```
   ahb_trans tr;
   virtual function void_build_ph();
      tr = ahb_trans::create_instance(this, "Ahb_Tr0",
         `__FILE__, `__LINE__);
      ...
   endfunction
endclass
```

## factory::override_with_new()

Sets the specified class instance as the create-by-construction `factory`.

### SystemVerilog

```
static function void classname::override_with_new(
   string name, classname factory, vmm_log log,
   string fname = "", int lineno = 0);
```

### Description

Sets the specified class instance as the create-by-construction factory, when creating further instances of that class under the specified instance name. You can specify the instance name as a match pattern or regular expression. Also, you can specify an instance name in a specific namespace by prefixing it with `spacename::`. The *`classname`*`::create_instance()` method uses the `allocate()` method to create a new instance of this class.

You should call this method using the following pattern:

```
master::override_with_new(
   "@*", extended_master::this_type(), this.log, `__FILE__,
   `__LINE__);
```

If the specified name matches the hierarchical name of atomic, single-stream, or multi-stream scenario generators of the appropriate type, then the matching factory instances they contain are immediately replaced with newly allocated instances of the specified class. If this method is called before the `build` phase, then this replacement is delayed until the completion of that phase.

The `log` argument is the message interface used by factory to report various messages. The `fname` and `lineno` arguments are used to track the file name and the line number where the instance is created using `create_instance`.

## Example

```
class my_ahb_trans extends vmm_object;
   `vmm_class_factory(my_ahb_trans)
endclass

initial begin
   ahb_trans::override_with_new("@%*",
      my_ahb_trans::this_type, log,
      `__FILE__, `__LINE__);
end
```

## factory::override_with_copy()

Schedules creation of a `factory` instance by copying the provided instance.

### SystemVerilog

```
static function void classname::override_with_copy(
   string name, classname factory, vmm_log log,
   string fname = "", int lineno = 0);
```

### Description

Sets the specified class instance as the create-by-copy `factory`, when creating further instances of that class under the specified instance name. You can specify the instance name as a match pattern or regular expression. Also, you can specify an instance name in a specific namespace by prefixing it with `spacename::`. The *`classname`*`::create_instance()` method uses the `copy()` method to create new instance of this class.

If the specified name matches the hierarchical name of atomic, single-stream, or multi-stream scenario generators of the appropriate type, the matching `factory` instances they contain are immediately replaced with copies of the specified `factory` instance. If you call this method before the `build` phase, this replacement is delayed until the completion of that phase.

The `log` argument is the message interface used by factory to report various messages. The `fname` and `lineno` arguments are used to track the file name and the line number where the instance is created using `create_instance`.

## Example

```
class ahb_trans extends vmm_object;
   rand bit [7:0] addr;
   `vmm_class_factory(ahb_trans)
endclass

initial begin
   ahb_trans tr;
   tr = new("gen_trans");
   tr.addr = 5;
   ahb_trans::override_with_copy("@%*", tr, log,
      `__FILE__, `__LINE__);
end
```

## factory::this_type()

Returns a dummy instance of the `factory` class.

### SystemVerilog

```
static function classname classname::this_type();
```

### Description

Returns a dummy instance of this class. You can use this class to call the *`classname`*`::allocate()` method.

### Example

```
ahb_trans::override_with_new("@%*",
my_ahb_trans::this_type,
   log, `__FILE__, `__LINE__);
```

## `vmm_class_factory(classname)

This is a macro for defining `factory` class.

### Description

Creates the `factory` class methods for the specified class. You must specify this method within the class declaration, with virtual `allocate()` and `copy()` methods. These virtual methods can be called without any arguments.

### Example

```
class ahb_trans extends vmm_object;
   rand bit [7:0] addr;
   ...
   `vmm_class_factory(ahb_trans)
endclass
```

# vmm_atomic_gen#(T)

Creates a parameterized version of the VMM atomic generator.

## SystemVerilog

```
class vmm_atomic_gen #(type T= vmm_data,
C=vmm_channel_typed#(T), string text = "") extends
vmm_atomic_gen_base;
```

## Description

The `vmm_atomic_generator macro creates a parameterized atomic generator. This generator can generate non-vmm_data transactions as well.

A macro is used to define a class named *class-name*_atomic_gen for any user-specified class derived from **vmm_data**[1], using a process similar to the **`vmm_channel** macro.

The atomic generator class is an extension of the **vmm_xactor** class and as such, inherits all the public interface elements provided in the base class.

## Example

```
class ahb_trans extends vmm_data;
   rand bit [31:0] addr;
   rand bit [31:0] data;
endclass

`vmm_channel(ahb_trans)
```

1. With a constructor callable without any arguments.

```
`vmm_atomic_gen(ahb_trans, "AHB Atomic Gen")
ahb_trans_channel chan0 = new("ahb_trans_chan", "chan0");
ahb_trans_atomic_gen   gen0  = new("AhbGen0", 0, chan0);
```

Is the same as:

```
vmm_channel_typed #(ahb_trans) chan0 = new("ahbchan",
   "chan0");
vmm_atomic_gen #(ahb_trans, , “AHB Atomic Gen”) gen0  =
new("AhbGen0", 0, chan0);
```

## Summary


- vmm_atomic_gen::<class-name>_channel out_chan ..... page A-15
- vmm_atomic_gen::enum {DONE} ..... page A-16
- vmm_atomic_gen::enum {GENERATED} ..... page A-17
- vmm_atomic_gen::inject() ..... page A-18
- vmm_atomic_gen::new() ..... page A-20
- vmm_atomic_gen::post_inst_gen() ..... page A-22
- vmm_atomic_gen::randomized_obj ..... page A-23
- vmm_atomic_gen::stop_after_n_insts ..... page A-25
- `vmm_atomic_gen() ..... page A-27
- `vmm_atomic_gen_using() ..... page A-28

## vmm_atomic_gen::<class-name>_channel out_chan

Reference the output channel for the instances generated by this transactor.

### SystemVerilog

```
class-name_channel out_chan;
```

### OpenVera

Not supported.

### Description

The output channel may have been specified through the constructor. If you did not specify any output channel instances, a new instance is automatically created. You may dynamically replace this reference in this property, but you should stop the generator during replacement.

### Example

*Example A-1*

```
program t( );
`vmm_atomic_gen(atm_cell, "ATM Cell")
   atm_cell_atomic_gen gen = new("Singleton");
   atm_cell cell;
   ...
   gen.out_chan.get(cell);
   ...
endprogram
```

## vmm_atomic_gen::enum {DONE}

Notification identifier for the notification service.

### SystemVerilog

```
enum {DONE};
```

### OpenVera

Not supported.

### Description

Notification identifier for the notification service that is in the **`vmm_xactor::notify`** property, provided by the **`vmm_xactor`** base class. It is configured as a **`vmm_xactor::ON_OFF`** notification, and is indicated when the generator stops, because the specified number of instances are generated. No status information is specified.

### Example

*Example A-2*

```
gen.notify.wait_for(atm_cell_atomic_gen::DONE);
```

## vmm_atomic_gen::enum {GENERATED}

Notification identifier for the notification service.

### SystemVerilog

```
enum {GENERATED};
```

### OpenVera

Not supported.

### Description

Notification identifier for the notification service interface that is in the **`vmm_xactor::notify`** property, provided by the **`vmm_xactor`** base class. It is configured as a **`vmm_xactor::ONE_SHOT`** notification, and is indicated immediately before an instance is added to the output channel. The generated instance is specified as the status of the notification.

### Example

*Example A-3*

```
gen.notify.wait_for(atm_cell_atomic_gen::GENERATED);
```

## vmm_atomic_gen::inject()

Injects the specified transaction or data descriptor in the output stream.

### SystemVerilog

```
virtual task inject(class-name data obj, ref bit dropped);
```

### OpenVera

Not supported.

### Description

You can use this method to inject directed stimulus, while the generator is running (with unpredictable timing) or when the generator is stopped.

Unlike injecting the descriptor directly in the output channel, it counts toward the number of instances generated by this generator and will be subjected to the callback methods. The method returns once the instance is consumed by the output channel or it is dropped by the callback methods.

### Example

*Example A-4*

```
task directed_stimulus;
  eth_frame to_phy, to_mac;
  ...
  to_phy = new;
  to_phy.randomize();
  ...
```

```
  fork
    env.host_src.inject(to_phy, dropped);
    begin
      // Force the earliest possible collision
      @ (posedge tb_top.mii.tx_en);
      env.phy_src.inject(to_mac, dropped);
    end
  join
  ...
  -> env.end_test;
endtask: directed_stimulus
```

## vmm_atomic_gen::new()

Creates a new instance of the *class-name*_**atomic_gen** class

### SystemVerilog

```
function new(string instance, int stream_id = -1,
class-name_channel out_chan = null, vmm_object parent =
null);
```

### OpenVera

Not supported.

### Description

Creates a new instance of the *class-name*_**atomic_gen** class, with the specified instance name and optional stream identifier. You can optionally connect the generator to the specified output channel. If you did not specify any output channel instance, one will be created internally in the *class-name*_**atomic_gen::out_chan** property.

The name of the transactor is defined as the user-defined class description string specified in the class implementation macro appended with “Atomic Generator”. The `parent` argument should be passed if implicit phasing needs to be enabled.

### Example

*Example A-5*

```
`vmm_channel(alu_data)
Class alu_env extends vmm_group;
   vmm_atomic_gen#(alu_data, ,"AluGen") gen_a;
```

```
   alu_data_channel alu_chan;
   . . .
   function void build_ph();
      alu_chan   = new ("ALU", "channel");
      gen_a      = new("Gen", 0,alu_chan ,this);
   endfunction
   . . .
endclass
```

## vmm_atomic_gen::post_inst_gen()

Invokes callback method, after a new transaction or data descriptor is created.

### SystemVerilog

```
virtual task post_inst_gen(class-name_atomic_gen gen,
     class-name obj, ref bit drop);
```

### OpenVera

Not supported.

### Description

The generator invokes the callback method, after a new transaction or data descriptor is created and randomized, but before it is added to the output channel.

The **`gen`** argument refers to the generator instance that is invoking the callback method (if the same callback extension instance is registered with more than one transactor instance). The **`data`** argument refers to the newly generated descriptor, which can be modified. If the value of the **`drop`** argument is set to non-zero, the generated descriptor will not be forwarded to the output channel. However, the remaining registered callbacks will be invoked.

## vmm_atomic_gen::randomized_obj

Randomizes the creation of random content of the output descriptor stream.

### SystemVerilog

```
class-name randomized_obj;
```

### OpenVera

Not supported.

### Description

Transaction or data descriptor instance that is repeatedly randomized to create the random content of the output descriptor stream. The individual instances of the output stream are copied from this instance, after randomization, using the **`vmm_data::copy()`** method.

The atomic generator uses a class factory pattern to generate the output stream instances. Using various techniques, you can constrain the generated stream on this property.

The **`vmm_data::stream_id`** property of this instance is set to the stream identifier of the generator, before each randomization. The **`vmm_data::data_id`** property of this instance is also set before each randomization. It will be reset to 0 when the generator is reset, and after the specified maximum number of instances are generated.

## Example

*Example A-6*

```
program test_...;
...
class long_eth_frame extends eth_frame;
  constraint long_frames {
    data.size() == max_len;
  }
endclass: long_eth_frame
...
initial begin
  env.build();
  begin
    long_eth_frame fr = new;
    env.host_src.randomized_obj = fr;
  end
  ...
  top.env.run();
end
endprogram
```

## vmm_atomic_gen::stop_after_n_insts

Stops, after the specified number of object instances are generated.

### SystemVerilog

```
int unsigned stop_after_n_insts;
```

### OpenVera

Not supported.

### Description

The generator stops, after the specified number of object instances are generated and consumed by the output channel. You must reset the generator, before it can be restarted. If the value of this property is 0, the generator will not stop on its own.

The default value of this property is 0.

### Example

*Example A-7*

```
program t( );
   `vmm_atomic_gen(atm_cell, "ATM Cell")
      atm_cell_atomic_gen gen = new("Singleton");
   gen.stop_after_n_insts = 10;
   ...
endprogram
```

# <*class-name*>_atomic_gen_callbacks

This class implements a façade for atomic generator, transactor, and callback methods. This class is automatically declared, and implemented for any user-specified class by the atomic generator macro.

## Summary


- `vmm_atomic_gen() ..... page A-27
- `vmm_atomic_gen_using() ..... page A-28

## ‘vmm_atomic_gen()

Defines an atomic generator class.

### SystemVerilog

```
‘vmm_atomic_gen(class-name, "Class Description")
```

### OpenVera

Not supported.

### Description

Defines an atomic generator class named *class-name*_**atomic_gen,** to generate instances of the specified class. The generated class must be derived from the **vmm_data** class, and the *class-name*_**channel** class must exist.

## ‘vmm_atomic_gen_using()

Defines an atomic generator class.

### SystemVerilog

```
‘vmm_atomic_gen_using(class-name, channel-type, "Class
Description")
```

### OpenVera

Not supported.

### Description

Defines an atomic generator class named *class-name*_**atomic_gen** to generate instances of the specified class, with the specified output channel type. The generated class must be compatible with the specified channel type, and both must exist.

You should use this macro, only while generating instances of a derived class that must be applied to a channel of the base class.

# vmm_atomic_scenario#(T)

Parameterized version of the VMM atomic scenario.

## SystemVerilog

```
class vmm_atomic_scenario #(T) extends vmm_ss_scenario#(T)
```

## Description

The parameterized atomic scenario is a generic typed scenario, extending `vmm_ss_scenario`. It is used by the parameterized scenario generator as the default scenario.

## Example

```
class ahb_trans extends vmm_data;
   rand bit [31:0] addr;
   rand bit [31:0] data;
endclass

`vmm_channel(ahb_trans)
`vmm_scenario_gen(ahb_trans, "AHB Scenario Gen")

class vmm_atomic_scenario#(ahb_trans) extends
   vmm_ss_scenario#(ahb_trans);
endclass
```

# vmm_broadcast

Channels are point-to-point data transfer mechanisms. If multiple consumers are extracting transaction descriptors from a channel, the transaction descriptors are distributed among various consumers and each of the *N* consumers view 1/*N* descriptors. If a point-to-multi-point mechanism is required, where all consumers must view all transaction descriptors in the stream, then a **`vmm_broadcast`** component can be used to replicate the stream of transaction descriptors from a source channel to an arbitrary and dynamic number of output channels. If only two output channels are required, the **`vmm_channel::tee()`** method of the source channel may also be used.

You can configure individual output channels to receive a copy of the reference to the source transaction descriptor (most efficient but the same descriptor instance is shared by the source and all like-configured output channels) or to use a new descriptor instance copied from the source object (least efficient but uses a separate instance that can be modified without affecting other channels or the original descriptor). A **`vmm_broadcast`** component can be configured to use references or copies in output channels by default.

In the *As Fast As Possible (AFAP)* mode, the full level of output channels is ignored. Only the full level of the source channel controls the flow of data through the broadcaster. Output channels are kept non-empty, as much as possible. As soon as an active output channel becomes empty, the next descriptor is removed from the source channel (if available), and added to all output channels, even if they are already full.

In the *As Late As Possible (ALAP)* mode, the slowest of the output or input channels controls the flow of data through the broadcaster. Only once, *all* active output channels are empty, the next descriptor is removed from the source channel (if available) and added to all output channels.

If there are no active output channels, the input channel is continuously drained as transaction descriptors are added to it to avoid data accumulation.

This class is based on the **`vmm_xactor`** class.

## Summary


- vmm_broadcast::add_to_output() ..... page A-32
- vmm_broadcast::bcast_off() ..... page A-34
- vmm_broadcast::bcast_on() ..... page A-35
- vmm_broadcast::broadcast_mode() ..... page A-36
- vmm_broadcast::log ..... page A-37
- vmm_broadcast::new() ..... page A-38
- vmm_broadcast::new_output() ..... page A-39
- vmm_broadcast::reset_xactor() ..... page A-40
- vmm_broadcast::set_input() ..... page A-41
- vmm_broadcast::start_xactor() ..... page A-42
- vmm_broadcast::stop_xactor() ..... page A-43

## vmm_broadcast::add_to_output()

Overloads to create broadcaster components with different broadcasting rules.

### SystemVerilog

```
virtual protected function bit
          add_to_output(int unsigned decision_id,
     int unsigned output_id,
     vmm_channel channel,
     vmm_data obj);
```

### OpenVera

Not supported.

### Description

Overloading this method, allows the creation of broadcaster components with different broadcasting rules. If this function returns TRUE (that is, non-zero), then the transaction descriptor will be added to the specified output channel. If this function returns FALSE (that is, zero), then the descriptor is not added to the channel. If the output channel is configured to use new descriptor instances, the **`obj`** parameter is a reference to that new instance.

This method is not necessarily invoked in increasing order of output identifiers. It is only called for output channels currently configured as ON. If this method returns FALSE for all output channels, for a given broadcasting cycle, lock-up may occur. The **`decision_id`** argument is reset to 0 at the start of every broadcasting cycle, and is incremented after each call to this method in the same cycle. It can be used to identify the start of broadcasting cycles.

If transaction descriptors are manually added to output channels, it is important that the **`vmm_channel::sneak()`** method be used to prevent the execution thread from blocking. It is also important that FALSE be returned to prevent that descriptor from being added to that output channel by the default broadcast operations, and thus from being duplicated into the output channel.

The default implementation of this method always returns TRUE.

## vmm_broadcast::bcast_off()

Turns broadcasting to the specified output channel off.

### SystemVerilog

```
virtual function void bcast_off(int unsigned output_id);
```

### OpenVera

Not supported.

### Description

By default, broadcasting to an output channel is on. When broadcasting is turned off, the output channel is flushed and the addition of new transaction descriptors from the source channel is inhibited. The addition of descriptors from the source channel is resumed, as soon as broadcasting is turned on.

If all output channels are off, the input channel is continuously drained to avoid data accumulation.

Any user extension of this method should call **`super.bcast_off()`**.

## vmm_broadcast::bcast_on()

Turns-on broadcasting to the specified output channel.

### SystemVerilog

```
virtual function void bcast_on(int unsigned output-id);
```

### OpenVera

Not supported.

### Description

By default, broadcasting to an output channel is on. When broadcasting is turned off, the output channel is flushed and the addition of new transaction descriptors from the source channel is inhibited. The addition of descriptors from the source channel is resumed, as soon as broadcasting is turned on.

If all output channels are off, the input channel is continuously drained to avoid data accumulation.

Any user extension of these methods should call **`super.bcast_on()`**.

## vmm_broadcast::broadcast_mode()

Changes the broadcasting mode to the specified mode.

### SystemVerilog

```
virtual function void broadcast_mode(bcast_mode_e mode);
```

### OpenVera

Not supported.

### Description

The new mode takes effect immediately. The available modes are specified by using one of the class-level enumerated symbolic values shown in Table A-1.

*Table A-1 Broadcasting Mode Enumerated Values*

*Table A-2*

| Enumerated Value | Broadcasting Operation |
|---|---|
| vmm_broadcast::ALAP | ***As Late As Possible***.<br>Data is broadcast *only* when all active output channels are empty. This delay ensures that data is not broadcast any faster than the slowest of all consumers can digest it. |
| vmm_broadcast::AFAP | As Fast As Possible.<br>Active output channels are kept non-empty, as much as possible. As soon as an active output channel becomes empty, the next descriptor from the input channel (if available) is immediately broadcast to all active output channels, regardless of their fill level<br><br>This mode *must not* be used if the data source can produce data at a higher rate than the slowest data consumer, and if broadcast data in all output channels are not consumed at the same average rate. |

## vmm_broadcast::log

Message service interface for the broadcast class.

### SystemVerilog

```
vmm_log log;
```

### OpenVera

Not supported.

### Description

Sets by the constructor, and uses the name and instance name specified in the constructor.

## vmm_broadcast::new()

Creates a new instance of a channel broadcaster object.

### SystemVerilog

```
function new(string name,
     string instance,
     vmm_channel source,
     bit use_references = 1,
     bcast_mode_typ mode = AFAP);
```

### OpenVera

Not supported.

### Description

Creates a new instance of a channel broadcaster object with the specified name, instance name, source channel, and broadcasting mode. If **`use_references`** is TRUE (that is, non-zero), references to the original source transaction descriptors are assigned to output channels by default (unless individual output channels are configured otherwise). The source can be assigned to null and set later by using “vmm_broadcast::set_input()” .

For more information on the available modes in the **`broadcast_mode()`** method, see the section “virtual function void broadcast_mode(bcast_mode_e mode);” on page 36.

### Example

*Example A-8*

```
vmm_broadcast bcast = new("Bcast", "", in_chan, 1);
```

## vmm_broadcast::new_output()

Adds the specified channel instance as a new output channel.

### SystemVerilog

```
virtual function int new_output(vmm_channel channel,
    logic use_references = 1'bx);
```

### OpenVera

Not supported.

### Description

Adds the specified channel instance as a new output channel to the broadcaster. If **`use_references`** is TRUE (that is, non-zero), references to the original source transaction descriptor is added to the output channel. If FALSE (that is, zero), a new instance copied from the original source descriptor is added to the output channel. If unknown (that is, 1'bx), the default broadcaster configuration is used.

If there are no output channels, the data from the input channel is continuously drained to avoid data accumulation.

This method returns a unique identifier for the output channel that must be used to modify the configuration of the output channel.

Any user extension of this method must call **`super.new_output()`**.

## vmm_broadcast::reset_xactor()

Resets this **`vmm_broadcast`** instance.

### SystemVerilog

```
virtual function void
  reset_xactor(reset_e rst_type = SOFT_RST);
```

### OpenVera

Not supported.

### Description

The broadcaster can be restarted. The input channel and all output channels are flushed.

## vmm_broadcast::set_input()

Specifies the channel as the source if not set previously

### System Verilog

```
function void set_input(vmm_channel source);
```

### Open Vera

Not supported

### Description

Identifies the channel as the source of the broadcaster, if the source is not set previously. If source is already set, then a warning is issued stating that this particular call has been ignored.

### Example

*Example A-9*

```
vmm_broadcast bcast = new("Bcast", "", null, 1);
bcast.set_input(in_chan);
```

## vmm_broadcast::start_xactor()

Starts this **`vmm_broadcast`** instance.

### SystemVerilog

```
virtual function void start_xactor();
```

### OpenVera

Not supported.

### Description

The broadcaster can be stopped. Any extension of this method must call **`super.start_xactor`**`()`.

### Example

*Example A-10*

```
vmm_broadcast bcast = new("Bcast", "", in_chan, 1);
bcast.start_xactor();
```

## vmm_broadcast::stop_xactor()

Suspends this **`vmm_broadcast`** instance.

### SystemVerilog

```
virtual function void stop_xactor();
```

### OpenVera

Not supported.

### Description

The broadcaster can be restarted. Any extension of this method must call **`super.stop_xactor()`**.

### Example

*Example A-11*

```
program test_directed;
...
initial begin
   ...
   env.start();
   env.host_src.stop_xactor();
   env.phy_src.stop_xactor();
   fork
      directed_stimulus;
   join_none
   env.run();
end
task directed_stimulus;
   ...
endtask: directed_stimulus
endprogram: test
```

# vmm_channel

This class implements a generic transaction-level interface mechanism.

Offset values, either accepted as arguments or returned values, are always interpreted the same way. A value of 0 indicates the head of the channel (first transaction descriptor added). A value of –1 indicates the tail of the channel (last transaction descriptor added). Positive offsets are interpreted from the head of the channel. Negative offsets are interpreted from the tail of the channel. For example, an offset value of –2 indicates the transaction descriptor just before the last transaction descriptor in the channel. It is illegal to specify a non-zero offset that does not correspond to a transaction descriptor, which is already in the channel.

The channel includes an active slot that can be used to create more complex transactor interfaces. The active slot counts toward the number of transaction descriptors currently in the channel, for control-flow purposes, but cannot be accessed nor specified through an offset specification.

The implementation uses a macro to define a class named *class-name*_**channel,** derived from the class named **vmm_channel,** for any user-specified class named *class-name*.

## Summary


- VMM Channel Relationships ..... page A-45
- VMM Channel Record or Replay ..... page A-47
- vmm_channel::activate() ..... page A-49
- vmm_channel::active_slot() ..... page A-51
- vmm_channel::connect() ..... page A-52
- vmm_channel::complete() ..... page A-54
- vmm_channel::empty_level() ..... page A-55
- vmm_channel::flow() ..... page A-56
- vmm_channel::flush() ..... page A-57

- vmm_channel::for_each() ..... page A-58
- vmm_channel::for_each_offset() ..... page A-59
- vmm_channel::full_level() ..... page A-60
- vmm_channel::get() ..... page A-61
- vmm_channel::get_consumer() ..... page A-62
- vmm_channel::get_producer() ..... page A-63
- vmm_channel::grab() ..... page A-64
- vmm_channel::level() ..... page A-66
- vmm_channel::is_full() ..... page A-67
- vmm_channel::is_grabbed() ..... page A-68
- vmm_channel::is_locked() ..... page A-70
- vmm_channel::kill() ..... page A-71
- vmm_channel::lock() ..... page A-72
- vmm_channel::log ..... page A-73
- vmm_channel::new() ..... page A-74
- vmm_channel::notify ..... page A-75
- vmm_channel::peek() ..... page A-77
- vmm_channel::playback() ..... page A-78
- vmm_channel::put() ..... page A-81
- vmm_channel::reconfigure() ..... page A-83
- vmm_channel::record() ..... page A-85
- vmm_channel::register_vmm_sb_ds() ..... page A-86
- vmm_channel::remove() ..... page A-87
- vmm_channel::set_consumer() ..... page A-88
- vmm_channel::set_producer() ..... page A-90
- vmm_channel::sink() ..... page A-92
- vmm_channel::size() ..... page A-93
- vmm_channel::sneak() ..... page A-94
- vmm_channel::start() ..... page A-96
- vmm_channel::status() ..... page A-97
- vmm_channel::tee() ..... page A-98
- vmm_channel::tee_mode() ..... page A-99
- vmm_channel::try_grab() ..... page A-100
- vmm_channel_typed#(type) ..... page A-102
- vmm_channel::ungrab() ..... page A-104
- vmm_channel::unlock() ..... page A-106
- vmm_channel::unput() ..... page A-107
- vmm_channel::unregister_vmm_sb_ds() ..... page A-108
- `vmm_channel() ..... page A-109


## VMM Channel Relationships

VMM extends its VMM channels, so that transactors acting as producer or consumer for this channel can be registered.

Hence, it is possible to verify that one unique producer or consumer pair is attached to a given channel. This insures that no collisions occur, even if you try to register new producer or consumer. In

addition, while registering channel producer or consumer, corresponding transactors are updated with input or output channels.

Using this class, you can avail benefits from built-in transactor uniqueness check and easily traverse transactor channels.

**`vmm_channel::set_producer()`** identifies the specified transactor as the current producer for the channel instance. This channel will be added to the list of output channels for the transactor. If a producer had been previously identified, the channel instance is removed from the list of previous producer output channels. Specifying a NULL transactor indicates that the channel does not contain any producer.

Although a channel can contain multiple producers (even though with an unpredictable ordering of each producer's contribution to the channel), only one transactor can be identified as the producer of a channel as they are primarily a point-to-point transaction-level connection mechanism.

**`vmm_channel::set_consumer()`** identifies the specified transactor as the current consumer for the channel instance. This channel will be added to the list of input channels for the transactor. If a consumer had been previously identified, the channel instance is removed from the list of previous consumer input channels. Specifying a NULL transactor indicates that the channel does not contain any consumer.

Although a channel can have multiple consumers (even though with an unpredictable distribution of input of each consumer from the channel), only one transactor can be identified as a consumer of a

channel as they are primarily a point-to-point transaction-level connection mechanism. The producer or consumer relationships are set from within transactors.

```
function xact::new(string inst,
                   tr_channel  in_chan = null,
                   obj_channel out_chan = null);
super.new("Xactor", inst);
   if (in_chan == null) in_chan = new(…);
   this.in_chan = in_chan;
   this.in_chan.set_consumer(this);
   if (out_chan == null) out_chan = new(…);
   this.out_chan = in_chan;
   this.out_chan.set_producer(this);
endfunction
```

**`vmm_channel::get_producer()`** — Returns the transactor that is specified as the current producer for the channel instance. Returns NULL, if no producer is identified.

**`vmm_channel::get_consumer()`** — Returns the transactor that is specified as the current consumer for the channel instance. Returns NULL, if no consumer is identified.

## VMM Channel Record or Replay

VMM extends its VMM channels so that incoming transactions can be stored to a file, and be replayed from this file later on.

It is possible to replay transactions either on-demand (for example, each time the channel is not blocking), or in a time-accurate way. With the time-accurate option, record or replay can replicate the original channel insertions scheme.

```
virtual task tb_env::start();
   ...
```

```
   if (vmm_opts::get_bit("record", "Record generator
output")) begin
      this.gen.out_chan.record("gen.dat");
   end
  if (vmm_opts::get_bit("play", "Playback recorded output"))
begin
      xaction tr = new;
      this.gen.out_chan.playback(ok, "gen.dat", tr);
   end
   else this.gen.start_xactor();
endtask
```

This feature is useful to speed-up time to debug by shutting down scenario generators. It can also be used to insure that the same data stream is always injected to channels.

```
class recorded_scenario extends vmm_ms_scenario;
  virtual task execute(ref int n);
    vmm_channel to_ahb = get_channel("ABUS");
    ahb_cycle tr = new;
    to_ahb.grab(this);
    fork
      forever begin: count
        to_ahb.notify.wait_for(vmm_channel::PUT);
        n++;
      end
    join_none
    to_ahb.playback(ok, "ahb.dat", tr, .grabber(this));
    to_ahb.release(this);
    disable count;
  endtask
endclass
```

## vmm_channel::activate()

Removes the transaction descriptor, which is currently in the active slot.

### SystemVerilog

```
task activate(output class-name obj, input int offset = 0);
```

### OpenVera

Not supported.

### Description

If the active slot is not empty, then this method first removes the transaction descriptor, which is currently in the active slot.

Move the transaction descriptor at the specified offset in the channel to the active slot ,and update the status of the active slot to **`vmm_channel::PENDING`**. If the channel is empty, then this method will wait until a transaction descriptor becomes available. The transaction descriptor is still considered as being in the channel.

It is an error to invoke this method with an offset value greater than the number of transaction descriptors currently in the channel, or to use this method with multiple concurrent consumer threads.

### Example

*Example A-12*

```
class consumer extends vmm_xactor;
   ...
   virtual task main();
```

```
      ...
      forever begin
         transaction tr;
         ...
         this.in_chan.activate(tr);
         this.in_chan.start();
         ...
         this.in_chan.complete();
         this.in_chan.remove();
      end
   endtask: main
   ...
endclass: consumer
```

## vmm_channel::active_slot()

Returns the transaction descriptor, which is currently in the active slot.

### SystemVerilog

```
function class-name active_slot();
```

### OpenVera

Not supported.

### Description

Returns the transaction descriptor, which is currently in the active slot. Returns *null,* if the active slot is empty.

## vmm_channel::connect()

Connects the output of this channel instance to the input of the specified channel instance.

### SystemVerilog

```
function void connect(vmm-channel downstream);
```

### OpenVera

Not supported.

### Description

The connection is performed with a blocking model to communicate the status of the downstream channel, to the producer interface of the upstream channel. Flushing this channel causes the downstream connected channel to be flushed as well. However, flushing the downstream channel does not flush this channel.

The effective full and empty levels of the combined channels is equal to the sum of their respective levels minus one. However, the detailed blocking behavior of various interface methods differ from using a single channel, with an equivalent configuration. Additional zero-delay simulation cycles may be required, while transaction descriptors are transferred from the upstream channel to the downstream channel.

Connected channels need not be of the same type, but must carry compatible polymorphic data.

The connection of a channel into another channel can be dynamically modified and broken by connection to a *null* reference. However, modifying the connection while there is data flowing through the channels may yield unpredictable behavior.

## vmm_channel::complete()

Updates the status of an active slot to **`vmm_channel::COMPLETED`**.

### SystemVerilog

```
function class-name complete(vmm_data status = null);
```

### OpenVera

Not supported.

### Description

The transaction descriptor remains in the active slot, and may be restarted. It is an error to call this method, if the active slot is empty. The **`vmm_data::ENDED`** notification of the transaction descriptor in the active slot is indicated with the optionally specified completion status descriptor.

### Example

*Example A-13*

```
class consumer extends vmm_xactor;
   virtual task main();
      forever begin
         transaction tr;
         this.in_chan.activate(tr);
         this.in_chan.start();
         this.in_chan.complete();
         this.in_chan.remove();
      end
   endtask: main
endclass: consumer
```

## vmm_channel::empty_level()

Returns the currently configured empty level.

### SystemVerilog

```
function int unsigned empty_level();
```

### OpenVera

Not supported.

## vmm_channel::flow()

Restores the normal flow of transaction descriptors through the channel.

### SystemVerilog

```
function void flow();
```

### OpenVera

Not supported.

## vmm_channel::flush()

Flushes the content of the channel.

### SystemVerilog

```
function void flush();
```

### OpenVera

Not supported.

### Description

Flushing unblocks any thread, which is currently blocked in the **`vmm_channel::put()`** method. This method causes the FULL notification to be reset, or the EMPTY notification to be indicated. Flushing a channel unlocks all sources and consumers.

## vmm_channel::for_each()

Iterates over all transaction descriptors, which are currently in the channel.

### SystemVerilog

```
function class-name for_each(bit reset = 0);
```

### OpenVera

Not supported.

### Description

The content of the active slot, if non-empty, is not included in the iteration. If the reset argument is TRUE, a reference to the first transaction descriptor in the channel is returned. Otherwise, a reference to the next transaction descriptor in the channel is returned. Returns *null,* when the last transaction descriptor in the channel is returned. It keeps returning *null,* unless reset.

Modifying the content of the channel in the middle of an iteration yields unexpected results.

## vmm_channel::for_each_offset()

Returns the offset of the last transaction descriptor, which is returned by the **`vmm_channel::for_each()`** method.

### SystemVerilog

```
function int unsigned for_each_offset();
```

### OpenVera

Not supported.

### Description

Returns the offset of the last transaction descriptor, which is returned by the **`vmm_channel::for_each()`** method. An offset of 0 indicates the first transaction descriptor in the channel.

## vmm_channel::full_level()

Returns the currently configured full level.

### SystemVerilog

```
function int unsigned full_level();
```

### OpenVera

Not supported.

## vmm_channel::get()

Retrieves the next transaction descriptor in the channel, at the specified offset.

### SystemVerilog

```
task get(output class-name obj, input int offset = 0);
```

### OpenVera

Not supported.

### Description

If the channel is empty, the function blocks until a transaction descriptor is available to be retrieved. This method may cause the **`EMPTY`** notification to be indicated, or the **`FULL`** notification to be reset. It is an error to invoke this method, with an offset value greater than the number of transaction descriptors, which are currently in the channel or with a non-empty active slot.

### Example

*Example A-14*

```
virtual function void build();
   fork
      forever begin
         eth_frame fr;
         this.mac.rx_chan.get(fr);
         this.sb.received_by_phy_side(fr);
       end
   join_none
endfunction: build
```

## vmm_channel::get_consumer()

Returns the current consumer for a channel.

### SystemVerilog

```
function vmm_xactor get_consumer();
```

### OpenVera

Not supported.

### Description

Returns the transactor that is specified as the current consumer for the channel instance. Returns `NULL`, if no consumer is identified.

### Example

*Example A-15*

```
class tr extends vmm_data;
endclass
`vmm_channel(tr)

class xactor extends vmm_xactor;
endclass

program prog;
   initial begin
      tr_atomic_gen agen = new("Atomic Gen");
      xactor xact = new("Xact", agen.out_chan);
      if (agen.out_chan.get_consumer() != xact) begin
       `vmm_error(log, "Wrong consumer for agen.out_chan");
      end
   end
endprogram
```

## vmm_channel::get_producer()

Returns the current producer for a channel.

### SystemVerilog

```
function vmm_xactor get_producer();
```

### OpenVera

Not supported.

### Description

Returns the transactor that is specified as the current producer, for the channel instance. Returns `NULL`, if no producer is identified.

### Example

*Example A-16*

```
class tr extends vmm_data;
endclass
`vmm_channel(tr)

class xactor extends vmm_xactor;
endclass

program prog;
   initial begin
      tr_atomic_gen agen = new("Atomic Gen");
      xactor xact = new("Xact", agen.out_chan);
      if (xact.in_chan.get_producer() != agen) begin
        `vmm_error(log, "Wrong producer for xact.in_chan");
      end
   end
endprogram
```

## vmm_channel::grab()

Grabs a channel for exclusive use.

### SystemVerilog

```
task grab(vmm_scenario grabber);
```

### OpenVera

```
task grab_t(rvm_scenario grabber);
```

### Description

Grabs a channel for the exclusive use of a scenario and its sub-scenarios. If the channel is currently grabbed by another scenario, the task blocks until the channel can be grabbed by the specified scenario descriptor. The channel will remain as grabbed, until it is released by calling the `vmm_channel::ungrab()` method.

If a channel is grabbed by a scenario that is a parent of the specified scenario, then the channel is immediately grabbed by the scenario.

If exclusive access to a channel is required outside of a scenario descriptor, then allocate a dummy scenario descriptor and use its reference.

When a channel is grabbed, the `vmm_channel::GRABBED` notification is indicated.

Note:Grabbing multiple channels creates a possible deadlock situation.

For example, two multi-stream scenarios may attempt to concurrently grab the same multiple channels, but in a different order. This may result in some of the channels to be grabbed by one of the scenario, and some of the channels to be grabbed by another scenario. This creates a deadlock situation, because neither scenario would eventually grab the remaining required channels.

## Example

*Example A-17*

```
class my_data extends vmm_data;
   ...
endclass
`vmm_channel(my_data)

class my_scenario extends vmm_ms_scenario;
   ...
endclass

program test_grab

   my_data_channel chan = new("Channel", "Grab", 10, 10);
   my_scenario scenario_1 = new;
   my_scenario scenario_2 = new;

   initial begin
      ...
      chan.grab(scenario_1);
      ...
      chan.ungrab(scenario_1);
      chan.grab(scenario_2);
      ...
   end

endprogram
```

## vmm_channel::level()

Returns the current fill level of the channel.

### SystemVerilog

```
function int unsigned level();
```

### OpenVera

Not supported.

### Description

The interpretation of the fill level depends on the configuration of the channel instance.

## vmm_channel::is_full()

Returns an indication of whether the channel is full or not.

### SystemVerilog

```
function bit is_full();
```

### OpenVera

Not supported.

### Description

Returns TRUE (that is, non-zero), if the fill level is greater than or equal to the currently configured full level. Otherwise, returns FALSE.

## vmm_channel::is_grabbed()

Checks if a channel is currently under exclusive use.

### SystemVerilog

```
function bit is_grabbed();
```

### OpenVera

```
function bit is_grabbed();
```

### Description

Returns `TRUE`, if the channel is currently grabbed by a scenario. Otherwise, returns `FALSE`.

### Example

*Example A-18*

```
class my_data extends vmm_data;
   ...
endclass
`vmm_channel(my_data)

class my_scenario extends vmm_ms_scenario;
   ...
endclass

program test_grab

   my_data_channel chan = new("Channel", "Grab", 10, 10);
   my_scenario scenario_1 = new;
   bit chan_status;

   initial begin
      ...
```

```
      chan_status = chan.is_grabbed();
      if(chan_status == 1)
        `vmm_note(log, "The channel is currently grabbed");
      else if(parent_grab == 0)
     `vmm_note(log, "The channel is currently not grabbed ");
      ...
   end

endprogram
```

## vmm_channel::is_locked()

Returns TRUE (non-zero), if any of the specified sides is locked.

### SystemVerilog

```
function bit is_locked(bit [1:0] who);
```

### OpenVera

Not supported.

### Description

Returns TRUE (non-zero), if any of the specified sides is locked. If both sides are specified, and if any side is locked, then returns TRUE.

### Example

*Example A-19*

```
while (chan.is_locked(vmm_channel::SOURCE
     vmm_channel::SINK))
   begin
      chan.notify.wait_for(vmm_channel::UNLOCKED);
   end
```

## vmm_channel::kill()

Prepares a channel for deletion.

### SystemVerilog

```
function void kill();
```

### OpenVera

Not supported.

### Description

Prepares a channel for deletion and reclamation by the garbage collector.

Remove this channel instance from the list of input and output channels of the transactors, which are identified as its producer and consumer.

### Example

*Example A-20*

```
program test_grab
   vmm_channel chan;

   initial begin
      chan = new("channel" ,"chan");
      ...
      chan.kill();
      ...
   end

endprogram
```

## vmm_channel::lock()

Blocks any source (consumer), as if the channel was full (empty), until explicitly unlocked.

### SystemVerilog

```
function void lock(bit [1:0] who);
```

### OpenVera

Not supported.

### Description

The side that is to be locked or unlocked is specified using the sum of the symbolic values, as shown in Table A-3.

Although the source and sink contain same control-flow effect, locking a source does not indicate the **`FULL`** notification, nor does locking the sink indicate the **`EMPTY`** notification.

*Table A-3 Channel Endpoint Identifiers*

*Table A-4*

| Symbolic Property | Channel Endpoint |
|---|---|
| vmm_channel::SOURCE | The producer side, i.e., any thread calling the **`vmm_channel::put()`** method |
| vmm_channel::SINK | The consumer side, i.e., any thread calling the **`vmm_channel::get()`** method |

## vmm_channel::log

Messages service interface for messages, issued from within the channel instance.

### SystemVerilog

```
vmm_log log;
```

### OpenVera

Not supported.

## vmm_channel::new()

Creates a new instance of a channel with the specified name, instance name, and full and empty levels.

### SystemVerilog

```
function new(string name,
             string instance,
             int unsigned full = 1,
             int unsigned empty = 0,
             bit fill_as_bytes = 0,
vmm_object parent = null);
```

### OpenVera

Not supported.

### Description

If the **`fill_as_bytes`** argument is TRUE (non-zero), then the full and empty levels and the fill level of the channel are interpreted as the number of bytes in the channel, as computed by the sum of **`vmm_data::byte_size()`** of all transaction descriptors in the channel and not the number of objects in the channel.

If the value is FALSE (zero), the full and empty levels, and the fill level of the channel are interpreted as the number of transaction descriptors in the channel.

It is illegal to configure a channel with a full level, lower than the empty level. The `parent` argument specifies the type of parent class which instantiates this channel.

## vmm_channel::notify

Indicates the occurrence of events in the channel.

### SystemVerilog

```
vmm_notify notify
```

### OpenVera

Not supported.

### Description

An event notification interface used to indicate the occurrence of significant events within the channel. The notifications shown in Table A-5 are pre-configured

*Table A-5 Pre-Configured Notifications in `vmm_channel` Notifier Interface*

| Symbolic Property | Corresponding Significant Event |
|---|---|
| vmm_channel::FULL | Channel is reached or surpassed its configured full level. This notification is configured as ON/OFF. Does not return any status. |
| vmm_channel::EMPTY | Channel is reached or underflowed the configured empty level. This event is configured as ON/OFF. Does not return any status. |
| vmm_channel::PUT | A new transaction descriptor is added to the channel. This event is configured as ONE_SHOT. The newly added transaction descriptor is available as status. |
| vmm_channel::GOT | A transaction descriptor is removed from the channel. This event is configured as ONE_SHOT. The newly removed transaction descriptor is available as status. |
| vmm_channel::PEEKED | A transaction descriptor is peeked from the channel. This event is configured as ONE_SHOT. The newly peeked transaction descriptor is available as status. |

| Symbolic Property | Corresponding Significant Event |
|---|---|
| vmm_channel:: ACTIVATED | A transaction descriptor is transferred to the active slot. This notification also implies a *PEEKED* notification. This event is configured as ONE_SHOT. The newly activated transaction descriptor is available as status. |
| vmm_channel:: ACT_STARTED | The state of a transaction descriptor in the active slot is updated to *STARTED*. This event is triggered ONE_SHOT. The currently active transaction descriptor is available as status. |
| vmm_channel:: ACT_COMPLETED | The state of a transaction descriptor in the active slot is updated to *COMPLETED*. This event is configured as ONE_SHOT. The currently active transaction descriptor is available as status. |
| vmm_channel:: ACT_REMOVED | A transaction descriptor is removed from the active slot. This notification also implies a *GOT* notification. This event is configured ONE_SHOT. The newly removed transaction descriptor is available as status. |
| vmm_channel::LOCKED | A side of the channel is locked. This event is configured as ONE_SHOT. |
| vmm_channel:: UNLOCKED | A side of the channel is unlocked. This event is configured as ONE_SHOT. |
| vmm_channel:: GRABBED | When a channel is grabbed, this notification is indicated. This event is configured as ONE_SHOT. |
| vmm_channel:: UNGRABBED | When a channel is ungrabbed, this notification is indicated. This event is configured as ONE_SHOT. |
| vmm_channel:: RECORDING | When the channel is being recorded, this notification is indicated. This event is configured as ON_OFF. |
| vmm_channel:: PLAYBACK | When the channel is being played, this notification is indicated. This event is configured as ON_OFF. |
| vmm_channel:: PLAYBACK_DONE | When the channel is being playback is done, this notification is indicated. This event is configured as ON_OFF. |

## vmm_channel::peek()

Gets a reference to the next transaction descriptor that will be retrieved from the channel, at the specified offset.

### SystemVerilog

```
task peek(output class-name obj, input int offset = 0);
```

### OpenVera

Not supported.

### Description

Gets a reference to the next transaction descriptor that will be retrieved from the channel, at the specified offset, without actually retrieving it. If the channel is empty, then the function will block until a transaction descriptor is available to be retrieved.

It is an error to invoke this method with an offset value greater than the number of transaction descriptors currently in the channel, or with a non-empty active slot.

### Example

*Example A-21*

```
class consumer extends vmm_xactor;
   virtual task main();
      forever begin
         transaction tr;
         this.in_chan.peek(tr);
         this.in_chan.get(tr);
      end
   endtask: main
endclass: consumer
```

## vmm_channel::playback()

Plays-back a recorded transaction stream.

### SystemVerilog

```
task playback(output bit success,
     input  string      filename,
     input  vmm_data    factory,
     input  bit         metered = 0,
     input  vmm_scenario grabber = null);
```

### OpenVera

```
task playback_t(var bit success,
     string      filename,
     rvm_data    factory,
     bit          metered = 0);
```

### Description

Injects the recorded transaction descriptors into the channel, in the same sequence in which they were recorded. The transaction descriptors are played back one-by-one, in the order found in the file. The recorded transaction stream replaces the producer for the channel. Playback does not need to happen in the same simulation run as recording. It can be executed in a different simulation run.

You must provide a non-null factory argument, of the same transaction descriptor type, as that with which recording was done. The `vmm_data::byte_unpack()` or `vmm_data::load()` method must be implemented for the transaction descriptor passed in to the `factory` argument.

If the `metered` argument is `TRUE`, then the transaction descriptors are played back (that is, sneak, put, or unput-ed) to the channel in the same relative simulation time interval, as the one in which they were originally recorded.

While playing back a recorded transaction descriptor stream on a channel, all other sources of the channel are blocked (for example, `vmm_channel::put()` from any other source be blocked). Transactions added using the `vmm_channel::sneak()` method would still be allowed from other sources, but a warning will be printed on any such attempt.

The `success` argument is set to `TRUE`, if the playback was successful. If the playback process encounters an error condition such as a `NULL` (empty string) filename, a corrupt file or an empty file, then `success` is set to `FALSE`.

When playback is completed, the `PLAYBACK_DONE` notification is indicated by `vmm_channel::notify`.

If the channel is currently grabbed by a scenario, other than the one specified, the playback operation will be blocked until the channel is ungrabbed.

## Example

*Example A-22*

```
class packet_env extends vmm_env;
   ...
   task start();
      ...
      `ifndef PLAY_DATA
         this.gen.start_xactor();
      `else
         fork
            begin
```

```
                bit success;
                data_packet factory = new;
                this.gen.out_chan.playback(success,
                                           "stimulus.dat",
                                           factory, 1);
                if (!this.success) begin
                   `vmm_error(this.log,
                              "Error during playback");
                end
             end
          join_none
       `endif
    endtask
    ...
endclass::packet_env
```

## vmm_channel::put()

Puts a transaction descriptor in the channel.

### SystemVerilog

```
task put(vmm_data obj,
     int offset = -1,
     vmm_scenario grabber = null);
```

### OpenVera

```
task put_t(rvm_data obj,
     integer offset = -1);
```

### Description

Adds the specified transaction descriptor to the channel. If the channel is already full, or becomes full after adding the transaction descriptor, then the task will block until the channel becomes empty.

If an offset is specified, then the transaction descriptor is inserted in the channel at the specified offset. An offset of 0 specifies at the head of the channel (LIFO order). An offset of -1 indicates the end of the channel (FIFO order).

If the channel is currently grabbed by a scenario other than the one specified, then this method will block and not insert the specified transaction descriptor in the channel, until the channel is ungrabbed or grabbed by the specified scenario.

### Example

*Example A-23*

```
class my_data extends vmm_data;
```

```
   ...
endclass
`vmm_channel(my_data)

class my_scenario extends vmm_ms_scenario;
   ...
endclass

program test_grab

   my_data_channel chan = new("Channel", "Grab", 10, 10);
   my_data md1 = new;
   my_scenario scenario_1 = new;

   initial begin
      ...
      chan.grab(scenario_1);
      chan.put(md1,scenario_1);
      ...
   end

endprogram
```

## vmm_channel::reconfigure()

Reconfigures the full or empty levels of the channel.

### SystemVerilog

```
function void reconfigure(int full = -1,
    int empty = -1,
    logic fill_as_bytes = 1'bx);
```

### OpenVera

Not supported.

### Description

If not negative, this method reconfigures the full or empty levels of the channel to the specified levels . Reconfiguration may cause threads, which are currently blocked on a **`vmm_channel::put()`** call to unblock. If the **`fill_as_bytes`** argument is specified as 1’b1 or 1’b0, then the interpretation of the fill level of the channel is modified accordingly. Any other value, leaves the interpretation of the fill level unchanged.

### Example

*Example A-24*

```
class consumer extends vmm_xactor;
   transaction_channel in_chan;
   ...
   function new(transaction_channel in_chan = null);
      ...
      if (in_chan == null)
        in_chan = new(...);
      in_chan.reconfigure(1);
```

```
        this.in_chan = in_chan;
    endfunction: new
    ...
endclass: consumer
```

## vmm_channel::record()

Starts recording the flow of transaction descriptors.

### SystemVerilog

```
function bit record(string filename);
```

### OpenVera

```
function bit record(string filename)
```

### Description

Starts recording the flow of transaction descriptors, which are added through the channel instance in the specified file. The **`vmm_data::save()`** method must be implemented for that transaction descriptor, and defines the file format. A transaction descriptor is recorded, when added to the channel by the **`vmm_channel::put()`** method.

A *`null`* filename stops the recording process. Returns TRUE, if the specified file was successfully opened.

## vmm_channel::register_vmm_sb_ds()

For more information, refer to the *VMM Scoreboard User Guide*.

## vmm_channel::remove()

Updates the status of the active slot to **vmm_channel::INACTIVE**.

### SystemVerilog

```
function class-name remove();
```

### OpenVera

Not supported.

### Description

Updates the status of the active slot to **vmm_channel::INACTIVE,** and removes the transaction descriptor from the active slot from the channel. This method may cause the **EMPTY** notification to be indicated, or the **FULL** notification to be reset. It is an error to call this method with an active slot in the **vmm_channel::STARTED** state. The **vmm_data::ENDED** notification of the transaction descriptor in the active slot is indicated.

### Example

*Example A-25*

```
class consumer extends vmm_xactor;
   virtual task main();
      forever begin
         transaction tr;
         this.in_chan.activate(tr);
         this.in_chan.start();
         this.in_chan.complete();
         this.in_chan.remove();
      end
   endtask: main
endclass: consumer
```

## vmm_channel::set_consumer()

Specifies the current consumer for a channel.

### SystemVerilog

```
function void set_consumer(vmm_xactor consumer);
```

### OpenVera

Not supported.

### Description

Identifies the specified transactor as the current consumer for the channel instance. This channel will be added to the list of input channels for the transactor. If a consumer is previously identified, the channel instance is removed from the previous list of consumer input channels.

Specifying a `NULL` transactor indicates that the channel does not contain any consumer.

Although a channel can contain multiple consumers (even though with unpredictable distribution of input of each consumer from the channel, only one transactor can be identified as a consumer of a channel, as they are primarily a point-to-point transaction-level connection mechanism.

### Example

*Example A-26*

```
class tr extends vmm_data;
 ...
```

```
endclass
`vmm_channel(tr)

class xactor extends vmm_xactor;
   ...
endclass

program prog;

   initial begin
      xactor xact = new("xact");
      tr_channel chan1 = new("tr_channel", "chan1");
      ...
      chan1.set_consumer(xact);
      ...
   end
endprogram
```

## vmm_channel::set_producer()

Specifies the current producer for a channel.

### SystemVerilog

```
function void set_producer(vmm_xactor producer);
```

### OpenVera

Not supported.

### Description

Identifies the specified transactor as the current producer for the channel instance. This channel will be added to the list of output channels for the transactor. If a producer is previously identified, the channel instance is removed from the previous list of producer output channels.

Specifying a `NULL` transactor indicates that the channel does not contain any producer.

Although a channel can have multiple producers (even though with unpredictable ordering of each contribution of a producer to the channel, only one transactor can be identified as a producer of a channel, as they are primarily a point-to-point transaction-level connection mechanism.

### Example

*Example A-27*

```
class tr extends vmm_data;
 ...
```

```
endclass
`vmm_channel(tr)
`vmm_scenario_gen(tr, "tr")

program prog;

   initial begin
      tr_scenario_gen sgen = new("Scen Gen");
      tr_channel chan1 = new("tr_channel", "chan1");
      ...
      chan1.set_producer(sgen);
      ...
   end
endprogram
```

## vmm_channel::sink()

Flushes the content of the channel, and sinks any further objects put into it.

### SystemVerilog

```
function void sink();
```

### OpenVera

Not supported.

### Description

No transaction descriptors will accumulate in the channel, while it is sunk. Any thread attempting to obtain a transaction descriptor from the channel will be blocked, until the flow through the channel is restored using the **`vmm_channel::flow()`** method. This method causes the FULL notification to be reset, or the EMPTY notification to be indicated.

## vmm_channel::size()

Returns the number of transaction descriptors, which are currently in the channel.

### SystemVerilog

```
function int unsigned size();
```

### OpenVera

Not supported.

### Description

Returns the number of transaction descriptors, which are currently in the channel, including the active slot, regardless of the interpretation of the fill level.

## vmm_channel::sneak()

Sneaks a transaction descriptor in the channel.

### SystemVerilog

```
function void sneak(vmm_data obj,
     int offset = -1,
     vmm_scenario grabber = null);
```

### OpenVera

```
task sneak(rvm_data obj,
     integer offset = -1);
```

### Description

Adds the specified transaction descriptor to the channel. This method will never block, even if the channel is full. An execution thread calling this method must contain some other throttling mechanism, to prevent an infinite loop from occurring.

This method is designed to be used in circumstances, where potentially blocking the execution thread could yield invalid results. For example, monitors must use this method to avoid missing observations.

If an offset is specified, the transaction descriptor is inserted in the channel at the specified offset. An offset of 0 specifies at the head of the channel (for example, LIFO order). An offset of -1 indicate the end of the channel (for example, FIFO order).

If the channel is currently grabbed by a scenario, other than the one specified, the transaction descriptor will not be inserted in the channel.

## Example

*Example A-28*

```
class my_data extends vmm_data;
   ...
endclass
`vmm_channel(my_data)

class my_scenario extends vmm_ms_scenario;
   ...
endclass

program test_grab

   my_data_channel chan = new("Channel", "Grab", 10, 10);
   my_data md1 = new;
   my_scenario scenario_1 = new;

   initial begin
      ...
      chan.grab(scenario_1);
      chan.sneak(md1,,scenario_1);
      ...
   end

endprogram
```

## vmm_channel::start()

Updates the status of the active slot to **`vmm_channel::STARTED`**.

### SystemVerilog

```
function class-name start();
```

### OpenVera

Not supported.

### Description

The transaction descriptor remains in the active slot. It is an error to call this method, if the active slot is empty. The **`vmm_data::STARTED`** notification of the transaction descriptor in the active slot is indicated.

### Example

*Example A-29*

```
class consumer extends vmm_xactor;
   ...
   virtual task main();
      forever begin
         transaction tr;
         ...
         this.in_chan.activate(tr);
         this.in_chan.start();
         ...
         this.in_chan.complete();
         this.in_chan.remove();
      end
   endtask: main
   ...
endclass: consumer
```

## vmm_channel::status()

Returns an enumerated value indicating the status of the transaction descriptor in the active slot.

### SystemVerilog

```
function active_status_e status();
```

### OpenVera

Not supported.

### Description

Returns one of the enumerated values, as shown in Table A-6, indicating the status of the transaction descriptor in the active slot.

*Table A-6 Pre-Configured Notifications in `vmm_channel` Notifier Interface*
*Table A-7*

| Symbolic Property | Corresponding Significant Event |
|---|---|
| `vmm_channel::INACTIVE` | No transaction descriptor is present in the active slot. |
| `vmm_channel::PENDING` | A transaction descriptor is present in the active slot, but it is not started yet. |
| `vmm_channel::STARTED` | A transaction descriptor is present in the active slot, and it is started, but it is not completed yet. The transaction is being processed by the downstream transactor. |
| `vmm_channel::COMPLETED` | A transaction descriptor is present in the active slot, and it is processed by the downstream transactor, but it is not yet removed from the active slot. |

## vmm_channel::tee()

Retrieves a copy of the transaction descriptor references that have been retrieved by the **`get()`** or **`activate()`** methods.

### SystemVerilog

```
task tee(output class-name obj);
```

### OpenVera

Not supported.

### Description

When the tee mode is ON, retrieves a copy of the transaction descriptor references that is retrieved by the **`get()`** or **`activate()`** methods. The task blocks until one of the **`get()`** or **`activate()`** methods successfully completes.

This method can be used to fork off a second stream of references to the transaction descriptor stream.

Note:The transaction descriptors themselves are not copied.

The references returned by this method are referring to the same transaction descriptor instances obtained by the **`get()`** and **`activate()`** methods.

## vmm_channel::tee_mode()

Turns the tee mode ON or OFF for this channel.

### SystemVerilog

```
function bit tee_mode(bit is_on);
```

### OpenVera

Not supported.

### Description

Returns TRUE, if the tee mode was previously ON. A thread that is blocked on a call to the **`vmm_channel::tee()`** method will not unblock execution, if the tee mode is turned OFF. If the stream of references is not drained through the **`vmm_channel::tee()`** method, data will accumulate in the secondary channel when the tee mode is ON.

## vmm_channel::try_grab()

Tries grabbing a channel for exclusive use.

### SystemVerilog

```
function bit try_grab(vmm_scenario grabber);
```

### OpenVera

```
function bit try_grab(rvm_scenario grabber);
```

### Description

Tries grabbing a channel for exclusive use and returns `TRUE`, if the channel was successfully grabbed by the scenario. Otherwise, it returns `FALSE`.

For more information on the channel grabbing rules, see the section `vmm_channel::grab()`.

### Example

*Example A-30*

```
class my_data extends vmm_data;
   ...
endclass
`vmm_channel(my_data)

class my_scenario extends vmm_ms_scenario;
   ...
endclass

program test_grab

   my_data_channel chan = new("Channel", "Grab", 10, 10);
```

```
   my_scenario scenario_1 = new;
   bit grab_success;

   initial begin
      ...
      grab_success = chan.try_grab(scenario_1);
      if(grab_success == 0)
         `vmm_error(log, "scenario_1 could not grab the
channel");
      else if(parent_grab == 1)
      `vmm_note(log, "scenario_1 has grabbed the channel ");
      ...
   end

endprogram
```

# vmm_channel_typed#(type)

Parameterized transaction-level interface.

## SystemVerilog

```
class vmm_channel_typed #(type T) extends vmm_channel;
```

## OpenVera

Not supported.

## Description

Parameterized class implementing a strongly typed transaction-level interface. The specified type parameter, *T* must be based on the `vmm_data` base class.

This class is the underlying class corresponding to the *`T`*`_channel` class that is created when using the `‘vmm_channel(T)` macro. They are both interchangeable. The parameterized class may be used directly, without having to declare the strongly-typed channel using the `‘vmm_channel()` macro beforehand.

The parameterized class also allows channels of parameterized classes to be defined without having to define an intermediate `typedef`.

## Example

*Example A-31 Equivalent definitions*

```
‘vmm_channel(eth_frame)
eth_frame_channel in_chan;
```

```
vmm_channel_typed#(eth_frame) in_chan;
```

*Example A-32 Equivalent definitions*

```
typedef apb_tr#(32, 64) apb_32_64_tr;
`vmm_channel(apb_32_64_tr) apb_32_64_tr_channel in_chan;
vmm_channel_typed#(apb_tr#(32, 64)) in_chan;
```

## vmm_channel::ungrab()

Releases a channel from exclusive use.

### SystemVerilog

```
function void ungrab(vmm_scenario grabber);
```

### OpenVera

```
task ungrab(rvm_scenario grabber);
```

### Description

Releases a channel that is previously grabbed for the exclusive use of a scenario, using the `vmm_channel::grab()` method. If another scenario is waiting to grab the channel, it will be immediately grabbed.

A channel must be explicitly ungrabbed, after the execution of an exclusive transaction stream is completed, to avoid creating deadlocks.

When a channel is ungrabbed, the `vmm_channel::UNGRABBED` notification is indicated.

### Example

*Example A-33*

```
class my_data extends vmm_data;
   ...
endclass
`vmm_channel(my_data)

class my_scenario extends vmm_ms_scenario;
```

```
   ...
endclass

program test_grab

   my_data_channel chan = new("Channel", "Grab", 10, 10);
   my_scenario scenario_1 = new;
   my_scenario scenario_2 = new;

   initial begin
      ...
      chan.grab(scenario_1);
      ...
      chan.ungrab(scenario_1);
      chan.grab(scenario_2);
      ...
   end

endprogram
```

## vmm_channel::unlock()

Blocks any source (consumer), as if the channel was full (empty), until explicitly unlocked.

### SystemVerilog

```
function void unlock(bit [1:0] who);
```

### OpenVera

Not supported.

### Description

The side that is to be locked or unlocked is specified using the sum of the symbolic values, as shown in Table A-3.

Although the source and sink contain the same control-flow effect, locking a source does not indicate the **`FULL`** notification, nor does locking the sink indicate the **`EMPTY`** notification.

## vmm_channel::unput()

Removes the specified transaction descriptor from the channel.

### SystemVerilog

```
function class-name unput(int offset = -1);
```

### OpenVera

Not supported.

### Description

It is an error to specify an offset to a transaction descriptor that does not exist.

This method may cause the **`EMPTY`** notification to be indicated, and causes the **`FULL`** notification to be reset.

## vmm_channel::unregister_vmm_sb_ds()

For more information, refer to the *VMM Scoreboard User Guide*.

## ‘vmm_channel()

Defines a channel class to transport instances of the specified class.

### SystemVerilog

`‘vmm_channel(`*`class-name`*`)`

### OpenVera

Not supported.

### Description

The transported class must be derived from the *vmm_data* class. This macro is typically invoked in the same file, where the specified class is defined and implemented.

This macro creates an external class declaration, and no implementation. It is typically invoked when the channel class must be visible to the compiler, but the actual channel class declaration is not yet available.

# vmm_connect#(T,N,D)

Utility class for connecting channels and notifications in the `vmm_unit::connect_ph()` method.

## SystemVerilog

```
class vmm_connect #(type T=vmm_channel, type N=T, type
D=vmm_data);
```

## Description

The `vmm_connect` utility class can be used for connecting channels and notifications in the `vmm_unit::connect_ph()` method. It performs additional check to verify whether the channels are already connected.

## Summary


- vmm_connect::channel() ..... page A-111
- vmm_connect::notify() ..... page A-112
- vmm_connect::tlm_bind() ..... page A-113
- vmm_connect::tlm_transport_interconnect() ..... page A-115

## vmm_connect::channel()

Connects the specified channel ports.

### SystemVerilog

```
class vmm_connect#(T)::channel(ref T upstream, downstream,
   string name = "", vmm_object parent = null);
```

### Description

Connects the specified channel ports (`upstream` and `downstream`). If both specified channels are not null, then they are connected using the `upstream.connect(downstream)` statement. Otherwise, both channels are connected by referring to the same channel instance. It is an error to attempt to connect two channels that are already connected together or to another channel. The optional argument `name` specifies the name of the binding, while `parent` is the component in which this binding is done.

### Example

```
class ahb_unit extends vmm_group;
   ahb_trans_channel gen_chan;
   ahb_trans_channel drv_chan;

   virtual function void build_ph();
      gen_chan = new("ahb_chan", "gen_chan");
      drv_chan = new("ahb_chan", "drv_chan");
   endfunction

   virtual function void connect_ph();
      vmm_connect#(ahb_trans_channel)::channel(
         gen_chan, drv_chan, "gen2drv", this);
   endfunction
endclass
```

## vmm_connect::notify()

Connects the specified observer to the specification notification.

### SystemVerilog

```
class vmm_connect#(T,N,D)::notify(N observer,
vmm_notify ntfy, int notification_id);
```

### Description

Connects the specified observer to the specification notification, using an instance of the `vmm_notify_observer` class. The specified argument `ntfy` indicates the notify class under which specified notification `notification_id` is registered. Each subsequent call to `ntfy.indicate(notification_id, tr)` allow to directly pass the transaction `tr` to the `observer`.

### Example

```
class scoreboard;
   virtual function void observe_trans(ahb_trans tr);
   endfunction
endclass
`vmm_notify_observer(scoreboard, observe_trans)

class ahb_unit extends vmm_group;
   scoreboard sb;
   virtual function void build_ph();
      sb = new();
   endfunction

   virtual function void connect_ph();
      vmm_connect#(.N(scoreboard), .D(ahb_trans))::notify(
         sb, mon.notify, mon.TRANS_STARTED);
   endfunction
endclass
```

## vmm_connect::tlm_bind()

Connects a VMM channel to a TLM interface.

### SystemVerilog

```
class vmm_connect#(.D(d))::tlm_bind(
         vmm_channel_typed#(D) channel ,
         vmm_tlm_base tlm_intf,
         vmm_tlm::intf_e intf,
         string fname = "", int lineno = 0);
```

### Description

Connects the specified VMM channel `channel` to the specified TLM interface `tlm_intf`. The TLM interface can be of any type as provided with `intf` such as `vmm_tlm::TLM_BLOCKING_PORT`, `vmm_tlm::TLM_BLOCKING_EXPORT`.

### Example

```
class Environment extends vmm_env;
  packet_atomic_gen  gen[];
  tlm_driver         drv[];

  virtual function void build_ph();
    gen = new[4];
    drv = new[4];
    for(int i=0; i<drv.size; i++) begin
      drv[i] = new($psprintf("Driver[%0d]", i), i, router);
     gen[i] = new($psprintf("Gen[%0d]", i), i);
    end
  endfunction

  virtual function void connect_ph();
    for(int i=0; i<drv.size; i++) begin
     vmm_connect #(.D(Packet))::tlm_bind(
         gen[i].out_chan,
```

```
            drv[i].socket,
            vmm_tlm::TLM_BLOCKING_PORT);
        end
    endfunction
endclass
```

## vmm_connect::tlm_transport_interconnect()

Connects TLM port to TLM export.

### SystemVerilog

```
static function tlm_transport_interconnect(vmm_tlm_base
tlm_intf_port, vmm_tlm_base tlm_intf_export,
vmm_tlm::intf_e intf, vmm_object parent = null, string fname
= "", int lineno = 0);
```

### Description

Binds the `tlm_intf_port` to `tlm_intf_export`, which are passed as arguments to the function.

First argument to the function is tlm port and the second argument is tlm export. If wrong types are passed to first or second argument then an error is issued.

Third argument takes the following values:

- `vmm_tlm::TLM_NONBLOCKING_EXPORT`

  This is used when producer is `vmm_tlm_b_transport_port` and consumer is `vmm_tlm_nb_transport_export.`

- `vmm_tlm::TLM_NONBLOCKING_FW_EXPORT`

  This is used when producer is `vmm_tlm_b_transport_port` and consumer is `vmm_tlm_nb_transport_fw_export`.

- `vmm_tlm::TLM_NONBLOCKING_PORT`

  This is used when producer is `vmm_tlm_nb_transport_port` and consumer is `vmm_tlm_b_transport_export`.

- `vmm_tlm::TLM_NONBLOCKING_FW_PORT`

  This is used when producer is `vmm_tlm_nb_transport_fw_port` and consumer is `vmm_tlm_b_transport_export`.

Any other values for third argument will issue an error.

## Example

```
class Environment extends vmm_env;
   tlm_gen gen;
   tlm_driver drv;

   virtual function void build_ph();
      gen = new(this,”tlm_gen”);
       drv = new(this,”tlm_driver”);
   endfunction

   virtual function void connect_ph();

vmm_connect#(vmm_channel,vmm_channel,my_trans)::tlm_transp
ort_interconnect(gen.socket,drv.socket,vmm_tlm::TLM_NONBLO
CKING_EXPORT,this);

   endfunction

endclass
```

# vmm_consensus

This class is used to determine when all the elements of a testcase, a verification environment, or a sub-environment agree that the test may be terminated.

## Summary


- vmm_consensus::consensus_force_thru() ..... page A-118
- vmm_consensus::forcing() ..... page A-119
- vmm_consensus::is_forced() ..... page A-120
- vmm_consensus::is_reached() ..... page A-121
- vmm_consensus::log ..... page A-122
- vmm_consensus::nays() ..... page A-123
- vmm_consensus::new() ..... page A-124
- vmm_consensus::notifications_e ..... page A-125
- vmm_consensus::psdisplay() ..... page A-126
- vmm_consensus::register_channel() ..... page A-127
- vmm_consensus::register_consensus() ..... page A-128
- vmm_consensus::register_no_notification() ..... page A-130
- vmm_consensus::register_notification() ..... page A-132
- vmm_consensus::register_voter() ..... page A-134
- vmm_consensus::register_xactor() ..... page A-136
- vmm_consensus::request() ..... page A-137
- vmm_consensus::unregister_channel() ..... page A-138
- vmm_consensus::unregister_consensus() ..... page A-139
- vmm_consensus::unregister_notification() ..... page A-140
- vmm_consensus::unregister_voter() ..... page A-142
- vmm_consensus::unregister_xactor() ..... page A-143
- vmm_consensus::wait_for_consensus() ..... page A-144
- vmm_consensus::wait_for_no_consensus() ..... page A-145
- vmm_consensus::yeas() ..... page A-146

## vmm_consensus::consensus_force_thru()

Forces sub-consensus through or not.

### SystemVerilog

```
function void consensus_force_thru(
    vmm_consensus vote,
    bit force_through = 1);
```

### OpenVera

```
Not supported
```

### Description

If the `force_through` argument is TRUE, any consensus forced on the specified sub-consensus instance will force the consensus on this `vmm_consensus` instance.

If the `force_through` argument is FALSE, any consensus forced on the specified sub-consensus instance will simply consent to the consensus on this `vmm_consensus` instance.

## vmm_consensus::forcing()

Returns a description of the forcing participants.

### SystemVerilog

```
function void forcing(ref string who[],
     ref string why[]);
```

### OpenVera

```
task forcing(var string who[*],
     var string why[*]);
```

### Description

Returns a description of the testbench elements that are currently forcing the end of test, and their respective reasons.

### Example

*Example A-34*

```
program test_consensus;

   string who[];
   string why[];
   vmm_consensus vote = new("Vote", "Main");

   initial begin
      ...
      vote.forcing(who,why);
      for(int i=0; i<who.size; i++)
         $display(" %s ------ %s",who[i],why[i]);
      ...
   end

endprogram
```

## vmm_consensus::is_forced()

Checks if a consensus is being forced.

### SystemVerilog

```
function bit is_forced();
```

### OpenVera

```
function bit is_forced();
```

### Description

This method returns an indication, if a participant forces a consensus. If the consensus is forced, a non-zero value is returned. If there is no consensus, or the consensus is not being forced, a zero value is returned.

### Example

*Example A-35*

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      ...
      if (vote.is_forced())
      `vmm_note (vote.log, "Consensus is forced");
      end
      ...
   end

endprogram
```

## vmm_consensus::is_reached()

Checks if a consensus is reached.

### SystemVerilog

```
function bit is_reached();
```

### OpenVera

```
function bit is_reached();
```

### Description

This method returns an indication, if a consensus is reached. If a consensus exists (whether forced or not), a non-zero value is returned. If there is no consensus, and the consensus is not being forced, a zero value is returned.

### Example

*Example A-36*

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      ...
      if (vote.is_reached())
      `vmm_note (vote.log, "Consensus is reached");
      else
      `vmm_error (vote.log, "Consensus has not reached");
      ...
   end

endprogram
```

## vmm_consensus::log

Message service interface for the consensus class.

### SystemVerilog

```
vmm_log log;
```

### OpenVera

```
rvm_log log;
```

### Description

This property is set by the constructor using the specified name and instance name. These names may be modified afterward, using the `vmm_log::set_name()` or `vmm_log::set_instance()` methods.

### Example

*Example A-37*

```
program test_consensus;
   vmm_consensus vote = new("Vote", "Main");

   initial begin
      ...
      if (vote.is_reached()) begin
      `vmm_note(vote.log, "Consensus has reached ");
      end else begin
      `vmm_note(vote.log, "Consensus has not reached yet");
      end
      ...
   end

endprogram
```

## vmm_consensus::nays()

Returns a description of the opposing participants.

### SystemVerilog

```
function void nays(ref string who[],
     ref string why[]);
```

### OpenVera

```
task nays(var string who[*],
     var string why[*]);
```

### Description

Returns a description of the testbench elements, which are currently opposing to the end of test, and their respective reasons.

### Example

*Example A-38*

```
program test_consensus;

   string who[];
   string why[];
   vmm_consensus vote = new("Vote", "Main");

   initial begin
      ...
      vote.nays(who,why);
      for(int i=0; i<who.size; i++)
         $display(" %s ------ %s",who[i],why[i]);
      ...
   end

endprogram
```

## vmm_consensus::new()

Creates a consensus, usually to determine the end-of-test.

### SystemVerilog

```
function new(string name, string inst, vmm_log log = null);
```

### OpenVera

```
task new(string name,
     string inst);
```

### Description

Creates a new instance of this class with the specified name and instance name. The specified name and instance names are used as the name and instance names of the log class property. You can pass a massage service interface(log) to consensus through constructor, if log is not being passed the it will create a new instance of log.

### Example

*Example A-39*

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      ...
   end
endprogram
```

## vmm_consensus::notifications_e

Predefined notifications.

### SystemVerilog

```
typedef enum int { NEW_VOTE = 999_999,
                   REACHED  = 999_998,
                   REQUEST  = 999_997} notifications_e;
```

### OpenVera

```
static integer NEW_VOTE;
```

### Description

Predefined notifications that are configured in `vmm_consensus::notify object`.

`NEW_VOTE` is a `ONE_SHOT` notification that is indicated whenever a participant changes its vote (using `vmm_consensus::consent`, `vmm_consensus::oppose` or `vmm_consensus::forced`). REACHED is a `ON_OFF` notification that is indicated whenever a test case end condition is reached or `unregister_all` method is called. REQUEST is a `ONE_SHOT` notification that is indicated whenever a request method is called.

## vmm_consensus::psdisplay()

Describes the status of the consensus.

### SystemVerilog

```
function string psdisplay(string prefix = "");
```

### OpenVera

```
function string psdisplay(string prefix = "");
```

### Description

Returns a human-readable description of the current status of the consensus, and who is opposing or forcing the consensus and why. Each line of the description is prefixed with the specified prefix.

### Example

*Example A-40*

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      ...
      $display(vote.psdisplay());
      ...
   end

endprogram
```

## vmm_consensus::register_channel()

Registers a channel as a participant.

### SystemVerilog

```
function void register_channel(vmm_channel chan);
```

### OpenVera

```
task register_channel(rvm_channel chan);
```

### Description

Adds a channel that can participate in this consensus. By default, a channel opposes the end of test if it is not empty, and consents to the end of test if it is currently empty. The channel may be later unregistered from the consensus using the "vmm_consensus::unregister_channel()" method.

### Example

*Example A-41*

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      vmm_channel v1 =new("Voter", "#1");
      ...
      vote.register_channel(v1);
      ...
   end

endprogram
```

## vmm_consensus::register_consensus()

Registers a sub-consensus as a participant.

### SystemVerilog

```
function void register_consensus(vmm_consensus vote,
bit force_through = 0);
```

### OpenVera

```
task register_consensus(vmm_consensus vote
     bit force_through = 0);
```

### Description

Adds a sub-consensus that can participate in this consensus. By default, a sub-consensus opposes the higher-level end of test if it is not reached its own consensus. Also, it consents to the higher-level end of test, if it is reached (or forced) its own consensus. The sub-consensus may be later unregistered from the consensus, using the "vmm_consensus::unregister_consensus()" method.

By default, a sub-consensus that has reached its consensus by force will not force a higher-level consensus, only consent to it. If the `force_through` parameter is specified as non-zero, a forced sub-consensus will force a higher-level consensus.

### Example

*Example A-42*

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");
```

```
   initial begin
      vmm_consensus c1;
      c1 = new("SubVote", "#1");
      ...
      vote.register_consensus(c1, 0);
      ...
   end

endprogram
```

## vmm_consensus::register_no_notification()

Registers a notification as a participant.

### SystemVerilog

```
function void register_no_notification(vmm_notify notify,
    int notification);
```

### OpenVera

```
task register_no_notification(rvm_notify notify,
    integer notification);
```

### Description

Adds an ON or OFF notification that can participate in this consensus. By default, a notification opposes the end of test if it is indicated, and consents to the end of test if it is not currently indicated. The notification may be later unregistered from the consensus using the "vmm_consensus::unregister_notification()" method.

For more information on the opposite polarity participation, see the section, "vmm_consensus::register_notification()" .

### Example

*Example A-43*

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      vmm_notify v1;
```

```
      vmm_log notify_log;
      notify_log = new ("Voter", "#1");
      v1 = new (notify_log);
      v1.configure(1, vmm_notify::ON_OFF);
      ...
      vote.register_no_notification(v1,1);
      ...
   end

endprogram
```

## vmm_consensus::register_notification()

Registesr a notification as a participant.

### SystemVerilog

```
function void register_notification(vmm_notify notify,
    int notification);
```

### OpenVera

```
task register_notification(rvm_notify notify,
    integer notification);
```

### Description

Adds an ON or OFF notification that can participate in this consensus. The specified argument `notify` is the handle of `vmm_notify` class under which specified notification is registered. By default, a notification opposes the end of test if it is not indicated, and consents to the end of test if it is currently indicated. The notification may be later unregistered from the consensus using the "vmm_consensus::unregister_notification()" method.

For more information on opposite polarity participation, see the "vmm_consensus::register_no_notification()" method.

### Example

*Example A-44*

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");
```

```
   initial begin
      vmm_notify v1;
      vmm_log notify_log;
      notify_log = new ("Voter", "#1");
      v1 = new (notify_log);
      v1.configure(1, vmm_notify::ON_OFF);
      ...
      vote.register_notification(v1,1);
      ...
   end

endprogram
```

## vmm_consensus::register_voter()

Registers a new general purpose participant.

### SystemVerilog

```
function vmm_voter register_voter(string name, vmm_voter
voter = null);
```

### OpenVera

```
function vmm_voter register_voter(string name);
```

### Description

Creates a new general-purpose voter interface that can participate in this consensus. The specified argument `name` indicates the name of a voter. If `null` is specified to the `vmm_voter` argument, an instance of `vmm_voter` will be internally allocated. By default, a voter opposes the end of test. The voter interface may be later unregistered from the consensus using the "vmm_consensus::unregister_voter()" method.

For more information on the general-purpose participant interface, see the section "vmm_voter" .

### Example

*Example A-45*

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      vmm_voter v1;
```

```
      ...
      v1 = vote.register_voter("Voter #1");
      ...
   end
endprogram
```

## vmm_consensus::register_xactor()

Registers a transactor as a participant.

### SystemVerilog

```
function void register_xactor(vmm_xactor xact);
```

### OpenVera

```
task register_xactor(rvm_xactor xact);
```

### Description

Adds a transactor that can participate in this consensus. A transactor opposes the end-of-test, if it is currently indicating the `vmm_xactor::IS_BUSY` notification. Moreover, it consents to the end of test, if it is currently indicating the `vmm_xactor::IS_IDLE` notification. The transactor may be later unregistered from the consensus using the "vmm_consensus::unregister_xactor()" method.

### Example

*Example A-46*

```
program test_consensus;
   vmm_consensus vote = new("Vote", "Main");

   initial begin
      vmm_xactor v1 =new("Voter", "#1");
      ...
      vote.register_xactor(v1);
      ...
   end
endprogram
```

## vmm_consensus::request()

Requests that a consensus be reached.

### SystemVerilog

```
task request(string why = "No reason specified",
vmm_consensus requester = null);
```

### OpenVera

```
Not supported
```

### Description

Makes a request of all currently-opposing participants, in this consensus instance that they consent to the consensus.

A request is made by indicating the `vmm_consensus::REQUEST` notification on the `vmm_consensus::notify` notification interface of this consensus instance, and all currently-opposing sub-consensus instances. If a request is made on a consensus instance that is a child of a `vmm_unit` instance, the `vmm_unit::consensus_requested()` method is also invoked.

If this consensus forces through to a higher-level consensus, the consensus request is propagated upward as well.

This task returns when the local consensus is reached.

## vmm_consensus::unregister_channel()

Unregisters a channel participant.

### SystemVerilog

```
function void unregister_channel(vmm_channel chan);
```

### OpenVera

```
task unregister_channel(rvm_channel chan);
```

### Description

Removes a previously registered channel from this consensus. If the channel was the only participant that objected to the consensus, the consensus will subsequently be reached.

### Example

*Example A-47*

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      vmm_channel v1 =new("Voter", "#1");
      ...
      vote.register_channel(v1);
      ...
      vote.unregister_channel(v1);
      ...
   end

endprogram
```

## vmm_consensus::unregister_consensus()

Unregisters a sub-consensus participant.

### SystemVerilog

```
function void unregister_consensus(vmm_consensus vote);
```

### OpenVera

```
task unregister_consensus(vmm_consensus vote);
```

### Description

Removes a previously registered sub-consensus from this consensus. If the sub-consensus was the only participant that objected to the consensus, the consensus will subsequently be reached. If the sub-consensus was forcing the consensus despite other objections, the consensus will subsequently no longer be reached.

### Example

*Example A-48*

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      vmm_consensus c1;
      c1 = new("SubVote", "#1");
      vote.register_consensus(c1, 0);
      ...
      vote.unregister_consensus(c1);
   end
endprogram
```

## vmm_consensus::unregister_notification()

Unregisters a notification participant.

### SystemVerilog

```
function void unregister_notification(vmm_notify notify,
    int notification);
```

### OpenVera

```
task unregister_notification(rvm_notify notify,
    integer notification);
```

### Description

Removes a previously registered ON or OFF notification from this consensus. The specified argument `notify` is the handle of `vmm_notify` class under which specified notification is registered. If the notification was the only participant that objected to the consensus, the consensus will subsequently be reached.

### Example

*Example A-49*

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      vmm_notify v1;
      vmm_log notify_log;
      notify_log = new ("Voter", "#1");
      v1 = new (notify_log);
      v1.configure(1, vmm_notify::ON_OFF);
      vote.register_notification(v1,1);
```

```
      vote.unregister_notification(v1,1);
   end
endprogram
```

## vmm_consensus::unregister_voter()

Unregisters a general purpose participant.

### SystemVerilog

```
function void unregister_voter(vmm_voter voter);
```

### OpenVera

```
task unregister_voter(vmm_voter voter);
```

### Description

Removes a previously registered general-purpose voter interface from this consensus. If the voter was the only participant that objected to the consensus, the consensus will subsequently be reached.

### Example

*Example A-50*

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      vmm_voter v1;
      ...
      v1 = vote.register_voter("Voter #1");
      ...
      vote.unregister_voter(v1);
      ...
   end

endprogram
```

## vmm_consensus::unregister_xactor()

Unregisters a transactor participant.

### SystemVerilog

```
function void unregister_xactor(vmm_xactor xact);
```

### OpenVera

```
task unregister_xactor(rvm_xactor xact);
```

### Description

Removes a previously registered transactor from this consensus. If the transactor was the only participant that objected to the consensus, then the consensus will subsequently be reached.

### Example

*Example A-51*

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      vmm_xactor v1 =new("Voter", "#1");
      ...
      vote.register_xactor(v1);
      ...
      vote.unregister_xactor(v1);
      ...
   end

endprogram
```

## vmm_consensus::wait_for_consensus()

Waits until a consensus is reached.

### SystemVerilog

```
task wait_for_consensus();
```

### OpenVera

```
task wait_for_consensus_t();
```

### Description

Waits until all participants, which explicitly consent and none oppose. There can be no abstentions.

If a consensus is already reached or forced, by the time this task is called, this task will return immediately.

A consensus may be broken later (if the simulation is still running) by any voter opposing the end of test, or a voter forcing the consensus deciding to consent normally or oppose normally.

### Example

*Example A-52*

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      vote.wait_for_consensus();
   end
endprogram
```

## vmm_consensus::wait_for_no_consensus()

Waits until a consensus is no longer reached.

### SystemVerilog

```
task wait_for_no_consensus();
```

### OpenVera

```
task wait_for_no_consensus_t();
```

### Description

Waits until a consensus is broken by no longer being forced and any one participant opposing. If a consensus is not reached, nor forced by the time this task is called, then this task will return immediately.

### Example

*Example A-53*

```
program test_consensus;

   vmm_consensus vote = new("Vote", "Main");

   initial begin
      ...
      vote.wait_for_no_consensus();
      ...
   end

endprogram
```

## vmm_consensus::yeas()

Returns a description of the consenting participants.

### SystemVerilog

```
function void yeas(ref string who[],
     ref string why[]);
```

### OpenVera

```
task yeas(var string who[*],
     var string why[*]);
```

### Description

Returns a description of the testbench elements currently consenting to the end of test, and their respective reasons.

### Example

*Example A-54*

```
program test_consensus;

   string who[];
   string why[];
   vmm_consensus vote = new("Vote", "Main");

   initial begin
      ...
      vote.yeas(who,why);
      for(int i=0; i<who.size; i++)
         $display(" %s ------ %s",who[i],why[i]);
      ...
   end

endprogram
```

# vmm_data

Models transactions efficiently.

## SystemVerilog

```
int vmm_data::lineno = 0
string vmm_data::filename = ""
```

## Description

The `lineno` and `filename` properties should be automatically set by the `create_instance()` method, and the predefined generators. Their content must be copied in the `vmm_data::copy_data()` method. If set to non-default values, their content should be displayed in the `vmm_data::psdisplay()` method.

Data modeling can be done more quickly due to unified data encapsulation, and by the presence of predefined methods for allocating, copying, comparing, displaying, and byte packing or unpacking of objects

This base class is to be used as the basis for all transaction descriptors and data models. It provides a standard set of methods expected to be found in all descriptors. It also creates a common class (similar to *`void`* type in C language) that can be used to create generic components.

The **`vmm_data`** class comes with shorthand macros that greatly facilitate data member declaration, and provide a quick way to implement the content of predefined methods. Implementing these methods provides an environment for other classes, such as **`vmm_channel`**, **`vmm_mss`**, **`vmm_scoreboard,`** and so on.

**The `vmm_data_member_begin()** method is used to start a shorthand section. The class name specified must be the name of the **vmm_data** extension class that is being implemented.

The shorthand section can only contain shorthand macros, and must be terminated by the **`vmm_data_member_end()** method, as shown in the following example.

```
class bus_trans extends vmm_data;
   typedef enum bit {READ=1'b0, WRITE=1'b1} kind_e;
   rand bit [31:0] addr;
   rand bit [31:0] data;
   rand kind_e     kind;
   `vmm_data_member_begin(bus_trans)
      `vmm_data_member_scalar(addr, DO_ALL)
      `vmm_data_member_scalar(data, DO_ALL)
      `vmm_data_member_enum(kind, DO_ALL)
   `vmm_data_member_end(bus_trans)
endclass
`vmm_channel(bus_trans)
`vmm_scenario_gen(bus_trans, "Gen")
```

The above example is for a simple transaction that contains no arrays. Note that appropriate macros should be used for arrays. Add the specified scalar type, fixed array of scalars, dynamic array of scalars, scalar-indexed associative array of scalars, or string-indexed associative array of scalars data member to the default implementation of the methods specified by the **do_what** argument.

```
class eth_frame extends vmm_data;
   vlan_frame vlan_fr_var[] ;
   ...
   `vmm_data_member_begin(eth_frame)
      `vmm_data_member_handle_da(vlan_fr_var, DO_ALL)
      ...
   `vmm_data_member_end(eth_frame)
   ...
endclass
```

**vmm_data::do_what_e** specifies which methods are to be provided by a shorthand implementation.

```
enum {DO_PRINT, DO_COPY, DO_COMPARE, DO_PACK, DO_UNPACK,
   DO_ALL} do_what_e;
```

It is used to specify which methods are to include the specified data members in their default implementation. Multiple methods can be specified by using **add** in the individual symbolic values. All methods are specified by specifying the **DO_ALL** symbol.

```
`vmm_data_member_scalar(len, DO_PRINT + DO_COPY +
   DO_COMPARE);
```

It is possible to override the default implementation of methods created by the **vmm_data** shorthand macros. **vmm_data::do_psdisplay()** overrides the shorthand **psdisplay()** method.

```
virtual function string do_psdisplay(string prefix = "")
```

This method overrides the default implementation of the **vmm_data::psdisplay()** method created by the **vmm_data** shorthand macros. If defined, it will be used instead of the default implementation.

The default implementation of this method in the **vmm_data** base class must not be called (for example, do not call **super.do_psdisplay()**).

The following are shorthand macros and the default implementations they replace:

| Shorthand Macro: | Overrides this default: |
|---|---|
| do_is_valid() | is_valid() |
| do_allocate() | allocate() |
| do_copy() | copy() |
| do_compare() | compare() |
| do_byte_size() | byte_size() |
| do_max_byte_size() | max_byte_size() |
| do_byte_pack() | byte_pack() |
| do_byte_unpack() | byte_unpack() |

## Summary


- vmm_data::byte_size() ..... page A-177
- vmm_data::byte_unpack() ..... page A-178
- vmm_data::do_byte_pack() ..... page A-180
- vmm_data::do_byte_pack() ..... page A-180
- vmm_data::do_byte_size() ..... page A-182
- vmm_data::do_byte_unpack() ..... page A-184
- vmm_data::do_compare() ..... page A-186
- vmm_data::do_copy() ..... page A-188
- vmm_data::do_how_e ..... page A-190
- vmm_data::do_is_valid() ..... page A-192
- vmm_data::do_max_byte_size() ..... page A-194
- vmm_data::do_psdisplay() ..... page A-195
- vmm_data::do_what_e ..... page A-196
- vmm_data::is_valid() ..... page A-197
- vmm_data::load() ..... page A-198
- vmm_data::set_log() ..... page A-199
- vmm_data::max_byte_size() ..... page A-200
- vmm_data::new() ..... page A-201
- vmm_data::notify ..... page A-202
- vmm_data::psdisplay() ..... page A-203
- vmm_data::save() ..... page A-204
- vmm_data::scenario_id ..... page A-205
- vmm_data::stream_id ..... page A-206

## ‘vmm_data_byte_size()

The shorthand implementation packing size methods.

### SystemVerilog

```
‘vmm_data_byte_size(max-expr, size-expr)
```

### OpenVera

Not supported.

### Description

Provides a default implementation of the `byte_size()` and `max_byte_size()` methods. The first and second expressions specify the value returned by the `max_byte_size()` and `byte_size()` methods respectively. The expression must be a valid SystemVerilog expression in the content of the class.

The shorthand implementation must be located immediately before the “‘vmm_data_member_begin()” .

### Example

*Example A-55*

```
class eth_frame extends vmm_data;
   ...
   ‘vmm_data_byte_size(1500, this.len_typ+16)
   ‘vmm_data_member_begin(eth_frame)
      ...
   ‘vmm_data_member_end(eth_frame)
...
endclass
```

## ‘vmm_data_member_begin()

Starts the shorthand section.

### SystemVerilog

```
‘vmm_data_member_begin(class-name)
```

### OpenVera

Not supported.

### Description

Starts the shorthand section providing a default implementation for the `psdisplay()`, `is_valid()`, `allocate()`, `copy()`, `compare()`, `byte_pack`, and `byte_unpack()` methods. A default implementation for the constructor is also provided, unless the “‘vmm_data_new()” method is previously specified.

In addition, a default implementation for `byte_size()` and `max_byte_size()` is also provided, unless the “‘vmm_data_byte_size()” method is previously specified.

The specified class-name must be the name of the `vmm_data` extension class that is being implemented.

The shorthand section can only contain shorthand macros, and must be terminated by the “‘vmm_data_member_end()” method.

### Example

*Example A-56*

```
class eth_frame extends vmm_data;
```

```
   ...
   `vmm_data_member_begin(eth_frame)
   `vmm_data_member_end(eth_frame)
endclass
```

## ‘vmm_data_member_end()

Terminates the shorthand section.

### SystemVerilog

```
‘vmm_data_member_end(class-name)
```

### OpenVera

Not supported.

### Description

Terminates the shorthand section providing a default implementation for the `psdisplay()`, `is_valid()`, `allocate()`, `copy()`, `compare()`, `byte_size()`, `max_byte_size()`, `byte_pack`, and `byte_unpack()` methods.

The class-name specified must be the name of the `vmm_data` extension class that is being implemented.

The shorthand section must have been started by the “`‘vmm_data_member_begin()`” method.

### Example

*Example A-57*

```
class eth_frame extends vmm_data;
   ...
   ‘vmm_data_member_begin(eth_frame)
      ...
   ‘vmm_data_member_end(eth_frame)
   ...
endclass
```

## ‘vmm_data_member_enum*()

The shorthand implementation for an enumerated data member.

### SystemVerilog

```
‘vmm_data_member_enum(member-name,
                      vmm_data::do_what_e do_what)

‘vmm_data_member_enum_array(member-name,
                            vmm_data::do_what_e do_what)

‘vmm_data_member_enum_da(member-name,
                         vmm_data::do_what_e do_what)

‘vmm_data_member_enum_aa_scalar(member-name,
                                vmm_data::do_what_e do_what)

‘vmm_data_member_enum_aa_string(member-name,
                                vmm_data::do_what_e do_what)
```

### OpenVera

Not supported.

### Description

Adds the specified enum-type, fixed array of enums, dynamic array of enums, scalar-indexed associative array of enums, or string-indexed associative array of enums data member to the default implementation of the methods, specified by the `do_what` argument.

The shorthand implementation must be located in a section started by “‘vmm_data_member_begin()” .

## Example

*Example A-58*

```
typedef enum bit[1:0] {NORMAL, VLAN, JUMBO } packet_type;

class eth_frame extends vmm_data;
   rand packet_type packet_type_var;
   ...
   `vmm_data_member_begin(eth_frame)
      `vmm_data_member_enum (packet_type_var, DO_ALL)
      ...
   `vmm_data_member_end(eth_frame)
   ...
endclass
```

## ‘vmm_data_member_handle*()

The shorthand implementation for a class handle data member.

### SystemVerilog

```
‘vmm_data_member_handle(member-name,
                        vmm_data::do_what_e do_what)

‘vmm_data_member_handle_array(member-name,
                              vmm_data::do_what_e do_what)

‘vmm_data_member_handle_da(member-name,
                           vmm_data::do_what_e do_what)

‘vmm_data_member_handle_aa_scalar(member-name,
                              vmm_data::do_what_e do_what)

‘vmm_data_member_handle_aa_string(member-name,
                              vmm_data::do_what_e do_what)
```

### OpenVera

Not supported.

### Description

Adds the specified handle-type fixed array of handles, dynamic array of handles, scalar-indexed associative array of handles, or string-indexed associative array of handles data member to the default implementation of the methods specified by the `do_what` argument.

The shorthand implementation must be located in a section started by `‘vmm_data_member_begin()`.

## Example

*Example A-59*

```
class vlan_frame;
   ...
endclass

class eth_frame extends vmm_data;
   vlan_frame vlan_fr_var ;
   ...
   `vmm_data_member_begin(eth_frame)
      `vmm_data_member_handle(vlan_fr_var, DO_ALL)
      ...
   `vmm_data_member_end(eth_frame)
   ...
endclass
```

## ‘vmm_data_new()

Starts the explicit constructor implementation.

### SystemVerilog

```
‘vmm_data_new(class-name)
```

### OpenVera

Not supported.

### Description

Specifies that an explicit user-defined constructor is used instead of the default constructor provided by the shorthand macros. Also, declares a “vmm_log” instance that can be passed to the base class constructor. Use this macro when data members must be explicitly initialized in the constructor.

The class-name specified must be the name of the vmm_data extension class that is being implemented.

This macro should be followed by the constructor declaration and must precede the shorthand data member section i.e., be located before the “‘vmm_data_member_begin()” macro.

### Example

*Example A-60*

```
class eth_frame extends vmm_data;
   ...
   ‘vmm_data_new(eth_frame)
      function new();
         super.new(this.log)
```

```
      ...
    endfunction

  `vmm_data_member_begin(eth_frame)
    ...
  `vmm_data_member_end(eth_frame)
  ...
endclass
```

## ‘vmm_data_member_scalar*()

The shorthand implementation for a scalar data member.

### SystemVerilog

```
‘vmm_data_member_scalar(member-name,
                        vmm_data::do_what_e do_what)

‘vmm_data_member_scalar_array(member-name,
                              vmm_data::do_what_e do_what)

‘vmm_data_member_scalar_da(member-name,
                           vmm_data::do_what_e do_what)

‘vmm_data_member_scalar_aa_scalar(member-name,
                              vmm_data::do_what_e do_what)

‘vmm_data_member_scalar_aa_string(member-name,
                              vmm_data::do_what_e do_what)
```

### OpenVera

Not supported.

### Description

Adds the specified scalar-type, fixed array of scalars, dynamic array of scalars, scalar-indexed associative array of scalars, or string-indexed associative array of scalars data member to the default implementation of the methods specified by the `do_what` argument.

A scalar is an integral type, such as bit, bit vector, and packed unions.

The shorthand implementation must be located in a section started by “‘vmm_data_member_begin()” .

## Example

*Example A-61*

```
class eth_frame extends vmm_data;
   rand bit [47:0] da;
   ...
   `vmm_data_member_begin(eth_frame)
      `vmm_data_member_scalar(da, DO_ALL);
      ...
   `vmm_data_member_end(eth_frame)
   ...
endclass
```

## ‘vmm_data_member_string*()

The shorthand implementation for a string data member.

### SystemVerilog

```
‘vmm_data_member_string(member-name,
                        vmm_data::do_what_e do_what)

‘vmm_data_member_string_array(member-name,
                              vmm_data::do_what_e do_what)

‘vmm_data_member_string_da(member-name,
                           vmm_data::do_what_e do_what)

‘vmm_data_member_string_aa_scalar(member-name,
                              vmm_data::do_what_e do_what)

‘vmm_data_member_string_aa_string(member-name,
                              vmm_data::do_what_e do_what)
```

### OpenVera

Not supported.

### Description

Adds the specified string-type, fixed array of strings, dynamic array of strings, scalar-indexed associative array of strings, or string-indexed associative array of strings data member to the default implementation of the methods specified by the `do_what` argument.

The shorthand implementation must be located in a section started by “‘vmm_data_member_begin()” .

## Example

*Example A-62*

```
class eth_frame extends vmm_data;
   string frame_name;
   ...
   `vmm_data_member_begin(eth_frame)
      `vmm_data_member_string(frame_name, DO_ALL)
      ...
   `vmm_data_member_end(eth_frame)
   ...
endclass
```

## ‘vmm_data_member_user_defined()

User-defined shorthand implementation data member.

### SystemVerilog

```
‘vmm_data_member_user_defined(member-name)
```

### OpenVera

Not supported.

### Description

Adds the specified user-defined default implementation of the member.

The shorthand implementation must be located in a section started by “‘vmm_data_member_begin()” .

### Example

*Example A-63*

```
class eth_frame extends vmm_data;
   rand bit [47:0] da;
   `vmm_data_member_begin(eth_frame)
      `vmm_data_member_user_defined(da)
   `vmm_data_member_end(eth_frame)
   function bit do_da ( input vmm_data::do_what_e do_what)

      do_da = 1;  // Success, abort by returning 0

      case (do_what)
      endcase
   endfunction
endclass
```

## ‘vmm_data_member_vmm_data*()

The shorthand implementation for a vmm_data-based data member.

### SystemVerilog

```
‘vmm_data_member_vmm_data(member-name,
                          vmm_data::do_what_e do_what,
                          vmm_data::do_how_e  do_how)

‘vmm_data_member_vmm_data_array(member-name,
                          vmm_data::do_what_e do_what,
                          vmm_data::do_how_e  do_how)

‘vmm_data_member_vmm_data_da(member-name,
                          vmm_data::do_what_e do_what,
                          vmm_data::do_how_e  do_how)

‘vmm_data_member_vmm_data_aa_scalar(member-name,
                          vmm_data::do_what_e do_what,
                          vmm_data::do_how_e  do_how)

‘vmm_data_member_vmm_data_aa_string(member-name,
                          vmm_data::do_what_e do_what,
                           vmm_data::do_how_e   do_how)
```

### OpenVera

Not supported.

### Description

Adds the specified vmm_data-type, fixed array of vmm_datas, dynamic array of vmm_datas, scalar-indexed associative array of vmm_datas, or string-indexed associative array of vmm_datas data member to the default implementation of the methods specified by the `do_what` argument. The `do_how` argument specifies whether the `vmm_data` values must be processed deeply or shallowly.

The shorthand implementation must be located in a section started by "`vmm_data_member_begin()" .

## Example

*Example A-64*

```
class vlan_frame extends vmm_data;
   ...
endclass

class eth_frame extends vmm_data;
   vlan_frame vlan_fr_var ;
   ...
   `vmm_data_member_begin(eth_frame)
    `vmm_data_member_vmm_data(vlan_fr_var, DO_ALL, DO_DEEP)
      ...
   `vmm_data_member_end(eth_frame)
   ...
endclass
```

## vmm_data::allocate()

Allocates a new instance.

### SystemVerilog

```
virtual function vmm_data allocate();
```

### OpenVera

Not supported.

### Description

Allocates a new instance of the same type as the object instance. Returns a reference to the new instance. Useful to implement class factories to create instances of user-defined derived class in generic code written using the base class type.

## vmm_data::compare()

Compares the current object instance with the specified object instance.

### SystemVerilog

```
virtual function bit compare(input vmm_data to,
     output string diff,
     input int kind = -1);
```

### OpenVera

Not supported.

### Description

Compares the current value of the object instance with the current value of the specified object instance, according to the specified kind. Returns TRUE (non-zero) if the value is identical. Returns FALSE, if the value is different, and a descriptive text of the first difference found is returned in the specified *`string`* variable. The **`kind`** argument may be used to implement different comparison functions (for example, full compare, comparison of *`rand`* properties only, comparison of all properties physically implemented in a protocol, and so on.)

### Example

*Example A-65*

```
function bit check(eth_frame actual)
   sb_where_to_find_frame where;
   eth_frame              q[$];
   eth_frame              expect;
```

```
    check = 0;
    if (!index_tbl[hash(actual)].exists()) return;
    where = index_tbl[hash(actual)];
    q = sb.port[where.port_no].queue[where.queue_no];
    expect = q.pop_front();
    if (actual.compare(expect)) check = 1;
endfunction: check
```

## vmm_data::copy()

Copies the current value of the object instance.

### SystemVerilog

```
virtual function vmm_data copy(vmm_data to = null);
```

### OpenVera

Not supported.

### Description

Copies the current value of the object instance to the specified object instance. If no target object instance is specified, a new instance is allocated. Returns a reference to the target instance.

### Example

*Example A-66*

The following trivial implementation does not work. Constructor copying is a shallow copy. The objects instantiated in the object (such as those referenced by the log and notify properties) are not copied, and both copies will share references to the same service interfaces. Moreover, it does not properly handle the case when the `to` argument is not null.

Invalid implementation of the **`vmm_data::copy()`** method:

```
function vmm_data atm_cell::copy(vmm_data to = null) copy =
new(this);
endfunction
```

The following implementation is usually preferable:

Proper implementation of the **`vmm_data::copy()`** method:

```
function vmm_data atm_cell::copy(vmm_data to = null)
   atm_cell cpy;

   if (to != null) begin
      if ($cast(cpy, to)) begin
         `vmm_fatal(log, "Not an atm_cell instance");
         return null;
      end
   end else cpy = new;

   this.copy_data(cpy);
   cpy.vpi = this.vpi;
   ...
   copy = cpy;
endfunction: copy
```

The base-class implementation of this method must not be called, as it contains error detection code of a derived class that forgot to supply an implementation. The **`vmm_data::copy_data()`** method should be called, instead.

## vmm_data::copy_data()

Copies the current value of all base class data properties.

### SystemVerilog

```
virtual protected function void copy_data(vmm_data to);
```

### OpenVera

Not supported.

### Description

Copies the current value of all base class data properties in the current data object, into the specified data object instance. This method should be called by the implementation of the **`vmm_data::copy()`** method, in classes immediately derived from this base class.

## vmm_data::data_id

Unique identifier for a data model or transaction descriptor instance.

### SystemVerilog

```
int data_id;
```

### OpenVera

Not supported.

### Description

Specifies the offset of the descriptor within a sequence, and the sequence offset within a stream. This property must be set by the transactor that instantiates the descriptor. It is set by the predefined generator, before randomization, so that it can be used to specify conditional constraints to express instance-specific or stream-specific constraints.

## vmm_data::display()

Displays the current value of the transaction or data.

### SystemVerilog

```
function void display(string prefix = "");
```

### OpenVera

Not supported.

### Description

Displays the current value of the transaction or data described by this instance, in a human-readable format on the standard output. Each line of the output will be prefixed with the specified prefix. This method prints the value returned by the **`psdisplay()`** method.

## vmm_data::byte_pack()

Packs the content of the transaction or data into a dynamic array of bytes.

### SystemVerilog

```
virtual function int unsigned byte_pack(
     ref logic [7:0] bytes[],
     input int unsigned offset = 0,
     input int kind = -1);
```

### OpenVera

Not supported.

### Description

Packs the content of the transaction or data into the specified dynamic array of bytes, starting at the specified offset in the array. The array is resized appropriately. Returns the number of bytes added to the array.

If the data can be interpreted or packed in different ways, the *kind* argument can be used to specify which interpretation or packing to use.

## vmm_data::byte_size()

Returns the number of bytes required to pack the content of this descriptor.

### SystemVerilog

```
virtual function int unsigned byte_size(int kind = -1);
```

### OpenVera

Not supported.

### Description

Returns the number of bytes required to pack the content of this descriptor. This method will be more efficient than the **`vmm_data::byte_pack()`** method, for knowing how many bytes are required by the descriptor, because no packing is actually done.

If the data can be interpreted or packed in different ways, the **`kind`** argument can be used to specify which interpretation or packing to use.

## vmm_data::byte_unpack()

Unpacks the specified number of bytes of data.

### SystemVerilog

```
virtual function int unsigned byte_unpack(
     const ref logic [7:0] bytes[],
     input int unsigned offset = 0,
     input int len = -1,
     input int kind = -1);
```

### OpenVera

Not supported.

### Description

Unpacks the specified number of bytes of data from the specified offset, in the specified dynamic array into this descriptor. If the number of bytes to unpack is specified as *-1*, the maximum number of bytes will be unpacked. Returns the number of bytes unpacked. If there is not enough data in the dynamic array to completely fill the descriptor, the remaining properties are set to unknown and a warning is generated.

If the data can be interpreted or unpacked in different ways, the **kind** argument can be used to specify which interpretation or packing to use.

### Example

*Example A-67*

```
class eth_frame extends vmm_data;
   ...
```

```
   typedef enum {UNTAGGED, TAGGED, CONTROL}
      frame_formats_e;
   rand frame_formats_e format;
   ...
   rand bit [47:0] dst;
   rand bit [47:0] src;
   rand bit        cfi;
   rand bit [ 2:0] user_priority;
   rand bit [11:0] vlan_id;
   ...
   virtual function int unsigned byte_unpack(
      const ref logic [7:0] array[],
      input int unsigned    offset = 0,
      input int             len    = -1,
      input int             kind   = -1);
      integer i;

      i = offset;
      this.format = UNTAGGED;
      ...
      if ({array[i], array[i+1]} === 16'h8100) begin
         this.format = TAGGED;
         i += 2;
         ...
         {this.user_priority, this.cfi, this.vlan_id} =
            {array[i], array[i+2]};
         i += 2;
         ...
      end
      ...
   endfunction: byte_unpack
   ...
endclass: eth_frame
```

## vmm_data::do_byte_pack()

Overrides the shorthand `byte_pack()` method.

### SystemVerilog

```
virtual int function do_byte_pack(ref logic [7:0] bytes[],
                                  input int unsigned offset = 0,
                                 input int kind = -1);
```

### OpenVera

Not supported.

### Description

This method overrides the default implementation of the `vmm_data::byte_pack()` method that is created by the `vmm_data` shorthand macros. If defined, this method is used instead of the default implementation.

The default implementation of this method in the `vmm_data` base class **must not** be called (for example, do not call `super.do_byte_pack()`).

The specified argument `bytes` is the dynamic array in which transaction contents are packed, starting at the specified `offset` value. The specified argument `kind` can be used to specify which interpretation or packing to use.

### Example

*Example A-68*

```
class eth_frame extends vmm_data;
```

```
   ...
   virtual int function do_byte_pack(ref logic [7:0]
        bytes[],input int unsigned offset = 0,
        input int kind = -1);
      int i;
      ...
      `ifdef ETH_USE_COMPOSITION
        {bytes[i], bytes[i+1]} = {this.vlan.user_priority,
            this.vlan.cfi, this.vlan.id};
      `else
        {bytes[i], bytes[i+1]} = {this.user_priority,
            this.cfi, this.vlan_id};
      `endif
      ...
   endfunction

endclass
```

## vmm_data::do_byte_size()

Overrides the shorthand `byte_size()` method.

### SystemVerilog

```
virtual int function do_byte_size(int kind  = -1);
```

### OpenVera

Not supported.

### Description

This method overrides the default implementation of the `vmm_data::byte_size()` method that is created by the `vmm_data` shorthand macros. If defined, this method is used instead of the default implementation.

The default implementation of this method in the `vmm_data` base class **must not** be called (for example, do not call `super.do_byte_size()`).

The returned value is the number of bytes required to pack the content of this descriptor. The specified `kind` argument can be used to specify which interpretation or packing to use.

### Example

*Example A-69*

```
class eth_frame extends vmm_data;
   virtual int function do_byte_size(int kind = -1);
      `ifdef TAGGED
     do_byte_size = 14 + data.size();
```

```
      `else
      do_byte_size = 14 + data.size() + 4;
      `endif
   endfunction
endclass
```

## vmm_data::do_byte_unpack()

Overrides the shorthand `byte_unpack()` method.

### SystemVerilog

```
virtual int function do_byte_unpack(
                            const ref logic [7:0] bytes[],
                            input int unsigned offset = 0,
                            input int len = -1,
                           input int kind = -1);
```

### OpenVera

Not supported.

### Description

This method overrides the default implementation of the `vmm_data::byte_unpack()` method that is created by the `vmm_data` shorthand macros. If defined, this method is used instead of the default implementation.

The default implementation of this method in the `vmm_data` base class **must not** be called (for example, do not call `super.do_byte_unpack()`).

The specified argument `len` is the number of data bytes to unpack, starting at specified `offset` value. The unpacked data is stored in the specified `bytes` dynamic array.

If the number of bytes to unpack is specified as -1, the maximum number of bytes will be unpacked. This method returns the number of bytes unpacked.

If the data can be interpreted or unpacked in different ways, the `kind` argument can be used to specify which interpretation or packing to used.

## Example

*Example A-70*

```
class eth_frame extends vmm_data;
   ...
  virtual int function do_byte_unpack(const ref logic [7:0]
      bytes[],input int unsigned offset = 0,
      input int len = -1,input int kind = -1);
     ...
     `ifdef ETH_USE_COMPOSITION
     {this.vlan.user_priority, this.vlan.cfi,
         this.vlan.id} = {bytes[i], bytes[i+1]};
     `else
     {this.user_priority, this.cfi, this.vlan_id} =
        {bytes[i], bytes[i+1]};
     `endif
     ...
   endfunction
   ...
endclass
```

## vmm_data::do_compare()

Overrides the shorthand **`compare()`** method.

### SystemVerilog

```
virtual bit function do_compare(input  vmm_data to,
                                output string diff,
                               input int kind=-1);
```

### OpenVera

Not supported.

### Description

This method overrides the default implementation of the `vmm_data::compare()` method that is created by the `vmm_data` shorthand macros. If defined, this method is used instead of the default implementation.

The default implementation of this method in the `vmm_data` base class **must not** be called (for example, do not call `super.do_compare()`).

The specified argument `to` is transaction instance with which current transaction is compared, returns TRUE if the value is identical. If the value is different, FALSE is returned and a descriptive text of the first difference found is returned in the specified string variable `diff`.

The `kind` argument may be used to implement different comparison functions (for example, full compare, comparison of `rand` properties only, comparison of all properties physically implemented in a protocol and so on.)

## Example

*Example A-71*

```
class eth_frame extends vmm_data;
   ...
   virtual bit function do_compare(input vmm_data to =
         null,output string diff, input int kind = -1);
      eth_frame fr;
      do_compare = 1;
      ...
      `ifdef ETH_USE_COMPOSITION
      if (fr.vlan == null) begin
          diff = "No vlan data";
        do_compare = 0;
       end

       if (fr.vlan.user_priority !==
            this.vlan.user_priority) begin
         $sformat(diff, "user_priority (3'd%0d !== 3'd%0d)",
                   this.vlan.user_priority,
                   fr.vlan.user_priority);
        do_compare = 0;
       end
      ...
      `else
      if (fr.user_priority !== this.user_priority) begin
         $sformat(diff, "user_priority (3'd%0d !== 3'd%0d)",
                   this.user_priority, fr.user_priority);
        do_compare = 0;
       end
      ...
      `endif
      ...
   endfunction
endclass
```

## vmm_data::do_copy()

Overrides the shorthand `copy()` method.

### SystemVerilog

```
virtual vmm_data function copy(vmm_data to = null);
```

### OpenVera

Not supported.

### Description

This method overrides the default implementation of the `vmm_data::copy()` method that is created by the `vmm_data` shorthand macros. If defined, this method is used instead of the default implementation.

The default implementation of this method in the `vmm_data` base class **must not** be called (for example, do not call `super.do_copy()`).

The optional `to` argument specifies the transaction on which copy needs to be performed.

### Example

*Example A-72*

```
class eth_frame extends vmm_data;
   ...
   virtual vmm_data function do_copy(vmm_data to = null);
      eth_frame cpy;
      if (to != null) begin
```

```
      if (!$cast(cpy, to)) begin
       `vmm_error(this.log, "Cannot copy to non-eth_frame\n
           instance");
       return null;
     end
     end else cpy = new;
     ...
     `ifdef ETH_USE_COMPOSITION
     if (this.vlan != null) begin
         cpy.vlan = new;
        cpy.vlan.user_priority = this.vlan.user_priority;
         cpy.vlan.cfi           = this.vlan.cfi;
         cpy.vlan.id            = this.vlan.id;
      end
     `else
       cpy.user_priority = this.user_priority;
       cpy.cfi           = this.cfi;
       cpy.vlan_id       = this.vlan_id;
     `endif
     ...
     do_copy = cpy;
   endfunction
   ...
endclass
```

## vmm_data::do_how_e

Specifies how `vmm_data` references are interpreted by a shorthand implementation.

### SystemVerilog

```
enum {      DO_NOCOPY      ='h001,
            DO_REFCOPY     ='h002,
            DO_DEEPCOPY    ='h004,
            HOW_TO_COPY    ='h007,
            DO_NOCOMPARE   ='h008,
            DO_REFCOMPARE  ='h010,
            DO_DEEPCOMPARE ='h020,
            HOW_TO_COMPARE ='h038,
            DO_NONE   ='h009,
            DO_REF    ='h012,
            DO_DEEP   ='h024,
            _DO_DUMMY} do_how_e;
```

### OpenVera

Not supported.

### Description

This method specifies how the copy and compare methods deal with a reference to a `vmm_data` instance, in their default implementation. Multiple mechanisms can be specified by using an `add` or an `or`, in the individual symbolic values. Following are the meanings of the `DO_NONE`, `DO_REF`, and `DO_DEEP` symbols:

- `DO_NONE` — Skips all comparison and copy operations
- `DO_REF` — Uses the reference itself, in comparison and copy operations

- `DO_DEEP` — Does deep compare and deep copy operations

## Example

*Example A-73*

```
‘vmm_data_member_vmm_data(parent, DO_ALL, DO_REF);
```

## vmm_data::do_is_valid()

Overrides the shorthand `is_valid()` method.

### SystemVerilog

```
virtual bit function do_is_valid(bit silent = 1,
                                 int kind  = -1);
```

### OpenVera

Not supported.

### Description

This method overrides the default implementation of the `vmm_data::is_valid()` method that is created by the `vmm_data` shorthand macros. If defined, this method is used instead of the default implementation. The default implementation of this method in the `vmm_data` base class **must not** be called (for example, do not call `super.do_is_valid()`).

If specified argument `silent` equals 1, no error or warning messages are issued if the content is invalid. If `silent` is FALSE, warning or error messages may be issued if the content is invalid. The meaning and use of the argument `kind` argument is descriptor-specific and defined by the user extension of this method.

### Example

*Example A-74*

```
class eth_frame extends vmm_data;
   virtual bit function do_is_valid(bit silent = 1,
         int kind = -1);
```

```
      do_is_valid = 1;
      if (!do_is_valid && !silent) begin
      `vmm_error(this.log, "Ethernet Frame is not valid");
      end
   endfunction
endclass
```

## vmm_data::do_max_byte_size()

Overrides the shorthand `max_byte_size()` method.

### SystemVerilog

```
virtual int function do_max_byte_size(int kind  = -1);
```

### OpenVera

Not supported.

### Description

This method overrides the default implementation of the `vmm_data::max_byte_size()` method that is created by the `vmm_data` shorthand macros. If defined, this method is used instead of the default implementation.

The default implementation of this method in the `vmm_data` base class **must not** be called (for example, do not call `super.do_max_byte_size()`).

### Example

*Example A-75*

```
class eth_frame extends vmm_data;
   virtual int function do_max_byte_size(int kind = -1);
      `ifdef JUMBO_PACKET
      do_max_byte_size = 9000;
      `else
      do_max_byte_size = 1500;
      `endif
   endfunction
endclass
```

## vmm_data::do_psdisplay()

Overrides the shorthand `psdisplay()` method.

### SystemVerilog

```
virtual function string do_psdisplay(string prefix = "")
```

### OpenVera

Not supported.

### Description

This method overrides the default implementation of the `vmm_data::psdisplay()` method that is created by the `vmm_data` shorthand macros. If defined, this method is used instead of the default implementation.

The default implementation of this method in the `vmm_data` base class **must not** be called (for example, do not call `super.do_psdisplay()`).

### Example

*Example A-76*

```
class eth_frame extends vmm_data;
   ...
  virtual function string do_psdisplay(string prefix = "")
    $sformat(psdisplay, "%sEthernet Frame #%0d.%0d.%0d:\n",
             prefix, this.stream_id, this.scenario_id,
             this.data_id);
      ...
   endfunction
endclass
```

## vmm_data::do_what_e

Specifies which methods are to be provided by a shorthand implementation.

### SystemVerilog

```
enum {DO_PRINT, DO_COPY, DO_COMPARE,
      DO_PACK, DO_UNPACK, DO_ALL} do_what_e;
```

### OpenVera

Not supported.

### Description

This method specifies which methods are to include the specified data members in their default implementation. Multiple methods can be specified by using an `add` or an `or`, in the individual symbolic values. All methods are specified by using the `DO_ALL` symbol.

### Example

*Example A-77*

```
`vmm_data_member_scalar(len,
                        DO_PRINT + DO_COPY + DO_COMPARE);
```

## vmm_data::is_valid()

Checks the current value of the transaction or data.

### SystemVerilog

```
virtual function bit is_valid(bit silent = 1,
     int kind = -1);
```

### OpenVera

Not supported.

### Description

Checks whether the current value of the transaction or data described by this instance is valid and error free, according to the optionally specified kind or format. Returns TRUE (non-zero), if the content of the object is valid. Otherwise, it returns FALSE. The meaning (and use) of the **`kind`** argument is descriptor-specific, and defined by the user extension of this method.

If **`silent`** is TRUE (non-zero), and if the content is invalid, then no error or warning messages are generated. If **`silent`** is FALSE, and if the content is invalid, then warning or error messages may be generated.

## vmm_data::load()

Sets the content of this descriptor.

### SystemVerilog

```
virtual function bit load(int file);
```

### OpenVera

Not supported.

### Description

Sets the content of this descriptor from the data, in the specified file. The format is user defined, and may be binary. By default, interprets a complete line as binary byte data and unpacks it.

Should return FALSE (zero), if the loading operation was not successful.

## vmm_data::set_log()

Replaces the message service interface for this instance of a data model or transaction descriptor.

### SystemVerilog

```
function vmm_log set_log(vmm_log log);
```

### OpenVera

Not supported.

### Description

Replaces the message service interface for this instance of a data model or transaction descriptor, with the specified message service interface. Also, it returns a reference to the previous message service interface. This method can be used to associate a descriptor with the message service interface of a transactor currently processing the transaction, or to set the service when it was not available during initial construction.

## vmm_data::max_byte_size()

Returns the maximum number of bytes required to pack the content of this descriptor.

### SystemVerilog

```
virtual function int unsigned max_byte_size(
     int kind = -1);
```

### OpenVera

Not supported.

### Description

Returns the maximum number of bytes, which are required to pack the content of any instance of this descriptor. A value of `0` indicates an unknown maximum size. Thsi method can be used to allocate memory buffers in the DUT or verification environment of suitable sizes.

If the data can be interpreted or packed in different ways, the *`kind`* argument can be used to specify which interpretation or packing to use.

## vmm_data::new()

Creates a new instance of this data model or transaction descriptor.

### SystemVerilog

```
function new(vmm_log log= null, vmm_object parent = null,
string name ="");
```

### OpenVera

Not supported.

### Description

Creates a new instance of this data model or transaction descriptor, with the specified message service interface. The specified message service interface is used when constructing the **`vmm_data::notify`** property.

### Example

*Example A-78*

Because of the potentially large number of instances of data objects, a *`class-static`* message service interface should be used to minimize memory usage and to control class-generic messages:

```
class eth_frame extends vmm_data {
   static vmm_log log = new("eth_frame", "class");
   function new();
      super.new(this.log);
      ...
   endfunction
endclass: eth_frame
```

## vmm_data::notify

A notification service interface with three pre-configured events.

### SystemVerilog

```
vmm_notify notify;
enum    {EXECUTE=999_999,
         STARTED=999_998,
         ENDED=999_997
        }notifications_e;
```

### OpenVera

Not supported.

### Description

The **`EXECUTE`** notification is ON or OFF, and indicated by default. It can be used to prevent the execution of a transaction or the transfer of data, if reset. The **`STARTED`** and **`ENDED`** notifications are ON or OFF events, and indicated by the transactor at the start and end of the transaction execution or data transfer. The meaning and timing of notifications is specific to the transactor, which is executing the transaction described by this instance.

## vmm_data::psdisplay()

Returns an image of the current value of the transaction or data.

### SystemVerilog

```
virtual function string psdisplay(string prefix = "");
```

### OpenVera

Not supported.

### Description

Returns an image of the current value of the transaction or data described by this instance, in a human-readable format as a string. The string may contain newline characters to split the image across multiple lines. Each line of the output must be prefixed with the specified prefix.

## vmm_data::save()

Appends the content of this descriptor to the specified file.

### SystemVerilog

```
virtual function void save(int file);
```

### OpenVera

Not supported.

### Description

Appends the content of this descriptor to the specified file. The format is user defined, and may be binary. By default, packs the descriptor and saves the value of the bytes, in sequence, as binary values and terminated by a newline.

## vmm_data::scenario_id

Unique identifier for a data model or transaction descriptor instance.

### SystemVerilog

```
int scenario_id;
```

### OpenVera

Not supported.

### Description

Specifies the offset of the descriptor within a sequence, and the sequence offset within a stream. This property must be set by the transactor that instantiates the descriptor. It is set by the predefined generator before randomization, so it can be used to specify conditional constraints to express instance-specific or stream-specific constraints.

## vmm_data::stream_id

Unique identifier for a data model or transaction descriptor instance.

### SystemVerilog

```
int stream_id;
```

### OpenVera

Not supported.

### Description

Specifies the offset of the descriptor within a sequence, and the sequence offset within a stream. This property must be set by the transactor that instantiates the descriptor. It is set by the predefined generator before randomization, so it can be used to specify conditional constraints to express instance-specific or stream-specific constraints.

# vmm_env

A base class used to implement verification environments.

## Summary


- vmm_env::build() ..... page A-208
- vmm_env::cfg_dut() ..... page A-209
- vmm_env::cleanup() ..... page A-210
- vmm_env::do_psdisplay() ..... page A-211
- vmm_env::do_start() ..... page A-212
- vmm_env::do_stop() ..... page A-213
- vmm_env::do_vote() ..... page A-214
- vmm_env::do_what_e ..... page A-215
- vmm_env::end_test ..... page A-216
- vmm_env::end_vote ..... page A-217
- vmm_env::gen_cfg() ..... page A-218
- vmm_env::log ..... page A-219
- vmm_env::new() ..... page A-220
- vmm_env::notify ..... page A-221
- vmm_env::report() ..... page A-222
- vmm_env::reset_dut() ..... page A-223
- vmm_env::run() ..... page A-224
- vmm_env::start() ..... page A-225
- vmm_env::stop() ..... page A-226
- vmm_env::wait_for_end() ..... page A-227
- `vmm_env_member_begin() ..... page A-228
- `vmm_env_member_channel*() ..... page A-229
- `vmm_env_member_end() ..... page A-231
- `vmm_env_member_enum*() ..... page A-232
- `vmm_env_member_scalar*() ..... page A-234
- `vmm_env_member_string*() ..... page A-236
- `vmm_env_member_subenv*() ..... page A-238
- `vmm_env_member_user_defined() ..... page A-240
- `vmm_env_member_vmm_data*() ..... page A-241
- `vmm_env_member_xactor*() ..... page A-243

## vmm_env::build()

Builds the verification environment.

### SystemVerilog

```
virtual function void build();
```

### OpenVera

Not supported.

### Description

Builds the verification environment, according to the value of the test configuration descriptor. If this method is not explicitly invoked in the test program, it will be implicitly invoked by the **`vmm_env::reset_dut()`** method.

### Example

*Example A-79*

```
class my_test extends vmm_test;
   ...
   virtual task run(vmm_env env);
      tb_env my_env;
      $cast(my_env, env);
      my_env.build();
      my_env.gen[0].start_xactor();
      my_env.run();
   endtask
endclass
```

## vmm_env::cfg_dut()

Configures the DUT.

### SystemVerilog

```
virtual task cfg_dut();
```

### OpenVera

Not supported.

### Description

Configures the DUT, according to the value of the test configuration descriptor. If this method is not explicitly invoked in the test program, it will be implicitly invoked by the **`vmm_env::start()`** method.

## vmm_env::cleanup()

Performs clean-up operations.

### SystemVerilog

```
virtual task cleanup();
```

### OpenVera

Not supported.

### Description

Performs clean-up operations to terminate the simulation, gracefully. Clean-up operations may include, letting the DUT drain off all buffered data, reading statistics registers in the DUT, and sweeping the scoreboard for leftover expected responses. If this method is not explicitly invoked in the test program, it will be implicitly invoked by the **`vmm_env::run()`** method.

## vmm_env::do_psdisplay()

Overrides the shorthand `psdisplay()` method.

### SystemVerilog

```
protected virtual function string do_psdisplay(string prefix
= "");
```

### OpenVera

Not supported.

### Description

This method overrides the default implementation of the `vmm_env::psdisplay()` method that is created by the `vmm_env` shorthand macros. If defined, this method is used instead of the default implementation.

### Example

*Example A-80*

```
class my_vmm_env extends vmm_env;
   ...

  virtual function string do_psdisplay(string prefix = "");
   $sformat(do_psdisplay,"%s Printing environment members",
        prefix);
      ...
   endfunction
   ...
endclass
```

## vmm_env::do_start()

Overrides the shorthand `start()` method.

### SystemVerilog

```
protected virtual task do_start()
```

### OpenVera

Not supported.

### Description

This method overrides the default implementation of the `vmm_env::start()` method that is created by the `vmm_env` shorthand macros. If defined, this method is used instead of the default implementation.

### Example

*Example A-81*

```
class my_vmm_env extends vmm_env;
   ...
   protected virtual task do_start();
      //vmm_env::start() operations
      ...
   endtask
   ...
endclass
```

## vmm_env::do_stop()

Overrides the shorthand `stop()` method.

### SystemVerilog

```
protected virtual task do_stop()
```

### OpenVera

Not supported.

### Description

This method overrides the default implementation of the `vmm_env::stop()` method that is created by the `vmm_env` shorthand macros. If defined, this method is used instead of the default implementation.

### Example

*Example A-82*

```
class my_vmm_env extends vmm_env;
   ...
   protected virtual task do_stop();
      //vmm_env::stop() operations
      ...
   endtask
   ...
endclass
```

## vmm_env::do_vote()

Overrides the shorthand voter registration.

### SystemVerilog

```
protected virtual task do_vote()
```

### OpenVera

Not supported.

### Description

This method overrides the default implementation of the voter registration that is created by the `vmm_env` shorthand macros. If defined, this method is used instead of the default implementation.

### Example

*Example A-83*

```
class my_vmm_env extends vmm_env;
   ...
   protected virtual task do_vote();
      //Register with this.end_vote
      ...
   endtask
   ...
endclass
```

## vmm_env::do_what_e

Specifies which methods should be provided by a shorthand implementation.

### SystemVerilog

```
enum {DO_PRINT, DO_START, DO_STOP, DO_RESET, DO_VOTE,
DO_ALL} do_what_e;
```

### OpenVera

Not supported.

### Description

Specifies which methods should include the specified data members in their default implementation. `"DO_PRINT"` includes the member in the default implementation of the `psdisplay()` method. `"DO_START"` includes the member in the default implementation of the `start()` method, if applicable. `"DO_STOP"` includes the member in the default implementation of the `stop()` method, if applicable. `"DO_VOTE"` automatically registers the member with the `vmm_env::end_vote` consensus instance, if applicable.

Multiple methods can be specified by adding or using an `or` in the individual symbolic values. All methods are specified by using the `"DO_ALL"` symbol.

### Example

*Example A-84*

```
`vmm_env_member_subenv(tcpip_stack, DO_ALL - DO_STOP);
```

## vmm_env::end_test

Causes the **`vmm_env::wait_for_end()`** method to return.

### SystemVerilog

```
event end_test;
```

### OpenVera

Not supported.

### Description

Causes the **`vmm_env::wait_for_end()`** method to return, when you trigger an event. It is up to the user-defined implementation of the **`vmm_env::wait_for_end()`** method to detect that this event is triggered and returned.

## vmm_env::end_vote

End-of-test consensus object.

### SystemVerilog

```
vmm_consensus end_vote;
```

### OpenVera

```
vmm_consensus end_vote;
```

### Description

Predefined end-of-test consensus instance that can be used in the extension of the `vmm_env::wait_for_end()` method, to determine that the simulation is reached its logical end. The name of the consensus is the name of the environment specified in the `vmm_env` constructor. The instance name of the consensus is "End-of-test Consensus".

Triggering the `vmm_env::end_test` event does not force the consensus. A consensus does not trigger the `end_test` event. This class property and the `end_test` event are not functionally related in the base class.

### Example

*Example A-85*

```
initial begin
   apb_env env;
  vmm_voter test_voter = env.end_vote.register_voter("Test
case Stimulus");
   ...
end
```

## vmm_env::gen_cfg()

Randomizes the test configuration descriptor.

### SystemVerilog

```
virtual function void gen_cfg();
```

### OpenVera

Not supported.

### Description

If this method is not explicitly invoked in the test program, it will be implicitly invoked by the **`vmm_env::build()`** method.

## vmm_env::log

Message service interface for the verification environment.

### SystemVerilog

```
vmm_log log;
```

### OpenVera

Not supported.

### Description

This property is set by the constructor, using the specified environment name, and may be modified at run time.

## vmm_env::new()

Creates an instance of the verification environment.

### SystemVerilog

```
function new(string name = "Verif Env");
```

### OpenVera

Not supported.

### Description

Creates an instance of the verification environment, with the specified name. The name is used as the name of the message service interface.

## vmm_env::notify

Notification service interface and predefined notifications.

### SystemVerilog

```
vmm_notify notify;
                      enum{GEN_CFG = 1,
                           BUILD,
                           RESET_DUT,
                           CFG_DUT,
                           START,
                           RESTART,
                           WAIT_FOR_END,
                           STOP,
                           CLEANUP,
                           REPORT,
                           RESTARTED} notifications_e;
```

### OpenVera

Not supported.

### Description

Notification service interface and predefined notifications used to indicate the progression of the verification environment. The predefined notifications are used to signal the start of the corresponding predefined virtual methods. All notifications are either ON or OFF.

## vmm_env::report()

Reports success or failure of the test, and closes all files.

### SystemVerilog

```
virtual task report();
```

### OpenVera

Not supported.

### Description

Reports final success or failure of the test, and closes all files. If this method is not explicitly invoked in the test program, it will be implicitly invoked by the **`vmm_env::run()`** method.

## vmm_env::reset_dut()

Resets the DUT to make it ready for configuration.

### SystemVerilog

```
virtual task reset_dut();
```

### OpenVera

Not supported.

### Description

Physically resets the DUT to make it ready for configuration. If this method is not explicitly invoked in the test program, it will be implicitly invoked by the **`vmm_env::cfg_dut()`** method.

## vmm_env::run()

Run the simulation.

### SystemVerilog

```
task run()
```

### OpenVera

Not supported.

### Description

Runs all remaining steps of the simulation, including **`vmm_env::stop()`**, **`vmm_env::cleanup(),`** and **`vmm_env::report()`**. This method must be explicitly invoked in the test programs.

## vmm_env::start()

Starts the test.

### SystemVerilog

```
virtual task start();
```

### OpenVera

Not supported.

### Description

Starts all the components of the verification environment to start the actual test. If this method is not explicitly invoked in the test program, it will be implictly invoked by the **`vmm_env::wait_for_end()`** method.

## vmm_env::stop()

Terminates the simulation, cleanly.

### SystemVerilog

```
virtual task stop();
```

### OpenVera

Not supported.

### Description

Terminates all components of the verification environment to terminate the simulation, cleanly. If this method is not explicitly invoked in the test program, it will be implicitly invoked by the **`vmm_env::cleanup()`** method.

## vmm_env::wait_for_end()

Waits for an indication that the test is reached completion.

### SystemVerilog

```
virtual task wait_for_end();
```

### OpenVera

Not supported.

### Description

Waits for an indication that the test is reached completion, or its objective. When this task returns, it signals that the end of simulation condition is detected. If this method is not explicitly invoked in the test program, it will be implicitly invoked by the **`vmm_env::stop()`** method.

### Example

*Example A-86*

```
class tb_env extends vmm_env;
   ...
   virtual task wait_for_end();
      super.wait_for_end();
      ...
      wait (this.cfg.run_for_n_tx_frames == 0 &&
            this.cfg.run_for_n_tx_frames == 0);
      ...
   endtask: wait_for_end
   ...
endclass: tb_env
```

## ‘vmm_env_member_begin()

Start the shorthand section.

### SystemVerilog

```
‘vmm_env_member_begin(class-name)
```

### OpenVera

Not supported.

### Description

Start the shorthand section, providing a default implementation for the `psdisplay()`, `start()` and `stop()` methods.

The class-name specified must be the name of the `vmm_env` extension class that is being implemented.

The shorthand section can only contain shorthand macros, and must be terminated by the “‘vmm_env_member_end()” method.

### Example

*Example A-87*

```
class tb_env extends vmm_env;
   ...
   ‘vmm_env_member_begin(tb_env)
      ...
   ‘vmm_env_member_end(tb_env)
   ...
endclass
```

## ‘vmm_env_member_channel*()

Shorthand implementation for a channel data member.

### SystemVerilog

```
‘vmm_env_member_channel(member-name,
                              vmm_env::do_what_e do_what)

‘vmm_env_member_channel_array(member-name,
                                   vmm_env::do_what_e do_what)

‘vmm_env_member_channel_aa_scalar(member-name,
                                   vmm_env::do_what_e do_what)

‘vmm_env_member_channel_aa_string(member-name,
                                   vmm_env::do_what_e do_what)
```

### OpenVera

Not supported.

### Description

Adds the specified channel-type, array of channels, dynamic array of channels, scalar-indexed associative array of channels, or string-indexed associative array of channels data member to the default implementation of the methods, specified by the `’do_what’` argument.

The shorthand implementation must be located in a section, which is started by the “‘vmm_env_member_begin()” method.

## Example

*Example A-88*

```
class my_vmm_env extends vmm_env;
   my_data_channel my_channel;
   ...
   `vmm_env_member_begin(my_vmm_env)
      `vmm_env_member_channel(my_channel,DO_ALL);
      ...
   `vmm_env_member_end(my_vmm_env)
   ...
endclass
```

## ‘vmm_env_member_end()

Terminates the shorthand section.

### SystemVerilog

```
‘vmm_env_member_end(class-name)
```

### OpenVera

Not supported.

### Description

Terminate the shorthand section, providing a default implementation for the `psdisplay()`, `start()`, and `stop()` methods.

The class-name specified must be the name of the `vmm_env` extension class that is being implemented.

The shorthand section must have been started by the “‘vmm_env_member_begin()” method.

### Example

*Example A-89*

```
class my_env extends vmm_env;
    ...
   `vmm_env_member_begin(my_vmm_env)
   `vmm_env_member_end(my_vmm_env)
    ...
endclass
```

## ‘vmm_env_member_enum*()

Shorthand implementation for an enumerated data member.

### SystemVerilog

```
‘vmm_env_member_enum(member-name,
                        vmm_env::do_what_e do_what)

‘vmm_env_member_enum_array(member-name,
                                vmm_env::do_what_e do_what)

‘vmm_env_member_enum_aa_scalar(member-name,
                                   vmm_env::do_what_e do_what)

‘vmm_env_member_enum_aa_string(member-name,
                                   vmm_env::do_what_e do_what)
```

### OpenVera

Not supported.

### Description

Adds the specified enum-type, array of enums, scalar-indexed associative array of enums, or string-indexed associative array of enums data member to the default implementation of the methods, specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by the “‘vmm_env_member_begin()” method.

### Example

*Example A-90*

```
typedef enum {blue,green,red,black} my_colors;
```

```
class my_vmm_env extends vmm_env;
   my_colors  color;
   ...
   `vmm_env_member_begin(my_vmm_env)
      `vmm_env_member_enum(color,DO_PRINT)
      ...
   `vmm_env_member_end(my_vmm_env)
   ...
endclass
```

## ‘vmm_env_member_scalar*()

Shorthand implementation for a scalar data member.

### SystemVerilog

```
‘vmm_env_member_scalar(member-name,
                       vmm_env::do_what_e do_what)

‘vmm_env_member_scalar_array(member-name,
                             vmm_env::do_what_e do_what)

‘vmm_env_member_scalar_aa_scalar(member-name,
                             vmm_env::do_what_e do_what)

‘vmm_env_member_scalar_aa_string(member-name,
                             vmm_env::do_what_e do_what)
```

### OpenVera

Not supported.

### Description

Add the specified scalar-type, array of scalars, scalar-indexed associative array of scalars or string-indexed associative array of scalars data member to the default implementation of the methods specified by the `’do_what’` argument.

A scalar is an integral type, such as bit, bit vector, and packed unions.

The shorthand implementation must be located in a section started by the “‘vmm_env_member_begin()” method.

## Example

*Example A-91*

```
class my_vmm_env extends vmm_env;
   bit [31:0] address;
   ...
   `vmm_env_member_begin(my_vmm_env)
      `vmm_env_member_scalar(address,DO_ALL)
      ...
   `vmm_env_member_end(my_vmm_env)
   ...
endclass
```

## ‘vmm_env_member_string*()

Shorthand implementation for a string data member.

### SystemVerilog

```
‘vmm_env_member_string(member-name,
                       vmm_env::do_what_e do_what)

‘vmm_env_member_string_array(member-name,
                             vmm_env::do_what_e do_what)

‘vmm_env_member_string_aa_scalar(member-name,
                                 vmm_env::do_what_e do_what)

‘vmm_env_member_string_aa_string(member-name,
                                 vmm_env::do_what_e do_what)
```

### OpenVera

Not supported.

### Description

Adds the specified string-type, array of strings, scalar-indexed associative array of strings, or string-indexed associative array of strings data member to the default implementation of the methods, specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by the “‘vmm_env_member_begin()” method.

### Example

*Example A-92*

```
class my_vmm_env extends vmm_env;
```

```
   string name;
   ...
   `vmm_env_member_begin(my_vmm_env)
      `vmm_env_member_string(name,DO_PRINT)
      ...
   `vmm_env_member_end(my_vmm_env)
   ...
endclass
```

## ‘vmm_env_member_subenv*()

Shorthand implementation for a transactor data member.

### SystemVerilog

```
‘vmm_env_member_subenv(member-name,
                              vmm_env::do_what_e do_what)

‘vmm_env_member_subenv_array(member-name,
                                   vmm_env::do_what_e do_what)

‘vmm_env_member_subenv_aa_scalar(member-name,
                                   vmm_env::do_what_e do_what)

‘vmm_env_member_subenv_aa_string(member-name,
                                   vmm_env::do_what_e do_what)
```

### OpenVera

Not supported.

### Description

Adds the specified sub-environment-type, array of sub-environments, dynamic array of sub-environments, scalar-indexed associative array of sub-environments, or string-indexed associative array of sub-environments data member to the default implementation of the methods, specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by the “‘vmm_env_member_begin()” method.

## Example

*Example A-93*

```
class my_subenv extends vmm_subenv
   ...
endclass

class my_vmm_env extends vmm_env;
   my_subenv  subenv ;
   ...
   `vmm_env_member_begin(my_vmm_env)
      `vmm_env_member_subenv(sub_env,DO_ALL);
      ...
   `vmm_env_member_end(my_vmm_env)
endclass
```

## ‘vmm_env_member_user_defined()

User-defined shorthand implementation data member.

### SystemVerilog

```
‘vmm_env_member_user_defined(member-name)
```

### OpenVera

Not supported.

### Description

Adds the specified user-defined default implementation of the methods, specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by the “`‘vmm_env_member_begin()”` method.

### Example

*Example A-94*

```
class my_vmm_env extends vmm_env;
   bit [7:0] env_id;
   ...
   `vmm_env_member_begin(my_vmm_env)
      `vmm_env_member_user_defined(env_id);
      ...
   `vmm_env_member_end(my_vmm_env)

   function bit do_env_id(vmm_env::do_what_e do_what)
      do_env_id = 1;
      case(do_what)
   endfunction
endclass
```

## ‘vmm_env_member_vmm_data*()

Shorthand implementation for a vmm_data-based data member.

### SystemVerilog

```
‘vmm_env_member_vmm_data(member-name,
                         vmm_env::do_what_e do_what)

‘vmm_env_member_vmm_data_array(member-name,
                         vmm_env::do_what_e do_what)

‘vmm_env_member_vmm_data_aa_scalar(member-name,
                         vmm_env::do_what_e do_what)

‘vmm_env_member_vmm_data_aa_string(member-name,
                         vmm_env::do_what_e do_what)
```

### OpenVera

Not supported.

### Description

Adds the specified vmm_data-type, array of `vmm_datas`, scalar-indexed associative array of `vmm_datas`, or string-indexed associative array of `vmm_datas` data member to the default implementation of the methods, specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by the `“‘vmm_env_member_begin()”` method.

## Example

*Example A-95*

```
class my_data extends vmm_data;
   ...
endclass : my_data

class my_vmm_env extends vmm_env;
   my_data   data;
   ...
   `vmm_env_member_begin(my_vmm_env)
      `vmm_env_member_vmm_data(data,DO_PRINT)
      ...
   `vmm_env_member_end(my_vmm_env)
   ...
endclass
```

## ‘vmm_env_member_xactor*()

Shorthand implementation for a transactor data member.

### SystemVerilog

```
‘vmm_env_member_xactor(member-name,
                              vmm_env::do_what_e do_what)

‘vmm_env_member_xactor_array(member-name,
                                   vmm_env::do_what_e do_what)

‘vmm_env_member_xactor_aa_scalar(member-name,
                                   vmm_env::do_what_e do_what)

‘vmm_env_member_xactor_aa_string(member-name,
                                   vmm_env::do_what_e do_what)
```

### OpenVera

Not supported.

### Description

Adds the specified transactor-type, array of transactors, dynamic array of transactors, scalar-indexed associative array of transactors, or string-indexed associative array of transactors data member to the default implementation of the methods, specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by the `“‘vmm_env_member_begin()”` method.

## Example

*Example A-96*

```
class my_data_gen extends vmm_xactor;
   ...
endclass

class my_vmm_env extends vmm_env;
   my_data_gen my_xactor;
   ...
   `vmm_env_member_begin(my_vmm_env)
      `vmm_env_member_xactor(my_xactor,DO_ALL);
      ...
   `vmm_env_member_end(my_vmm_env)
   ...
endclass
```

# vmm_group

Class to create structural elements.

## SystemVerilog

```
virtual class vmm_group extends vmm_unit;
```

## Description

This class is used as the base composition class for building structural elements composed of transactors or other groups.

This class can be a leaf or a non-leaf component.

## Example

```
class vip1 extends vmm_group;
endclass
```

## Summary


- vmm_group::new() ..... page A-246

## vmm_group::new()

Acts as a constructor for `vmm_group`.

### SystemVerilog

```
function vmm_group::new(string name =””, string inst =””,
   vmm_object parent = null);
```

### Description

Constructs an instance of this class with the specified name, instance name, and an optional parent.

The specified name is used as the name of the embedded `vmm_log`.

The specified instance name is used as the name of the underlying `vmm_object`.

### Example

```
class vip1 extends vmm_group;
   function new (string name, string inst);
      super.new (this,inst);
   endfunction
endclass
```

# vmm_group_callbacks

Facade class for callback methods provided by the `vmm_group`.

## Example

```
class group_callbacks extends vmm_group_callbacks;
   virtual function void my_f1();
   endfunction
   virtual function void my_f2();
   endfunction
endclass
```

## Summary


- vmm_group::append_callback() ..... page A-248
- vmm_group::prepend_callback() ..... page A-250
- vmm_group::unregister_callback() ..... page A-252

## vmm_group::append_callback()

Appends the specified callback.

### SystemVerilog

```
function void vmm_group::append_callback(
vmm_group_callbacks cb)
```

### Description

Appends the specified callback extension `cb` to the callback registry for this group.

### Example

```
class group_callbacks extends vmm_group_callbacks;
   virtual function void my_f1();
   endfunction
endclass
class groupExtension extends vmm_group;
   function new (string name, string inst,
         vmm_unit parent=null);
      super.new(name,inst,parent);
   endfunction
   function void build_ph();
      `vmm_callback(group_callbacks,my_f1());
   endfunction:build_ph
   ...
endclass
class groupExtension_callbacks extends group_callbacks;
   int my_f1_counter++;
   virtual function void my_f1();
      my_f1_counter++;
   endfunction
endclass
initial begin
   groupExtension g1 = new ("my_group", "g1");
```

```
   groupExtension_callbacks cb1 = new();
   g1.append_callback(cb1);
   ...
end
```

## vmm_group::prepend_callback()

Prepends the specified callback.

### SystemVerilog

```
function void vmm_group::prepend_callback(
vmm_group_callbacks cb)
```

### Description

Prepends the specified callback extension `cb` to the callback registry, for this group.

### Example

```
class group_callbacks extends vmm_group_callbacks;
   virtual function void my_f1();
   endfunction
endclass
class groupExtension extends vmm_group;
   function new (string name, string inst,
         vmm_unit parent=null);
      super.new(name,inst,parent);
   endfunction
   function void build_ph();
      `vmm_callback(group_callbacks,my_f1());
   endfunction:build_ph
   ...
endclass

class groupExtension_callbacks extends group_callbacks;
   int my_f1_counter++;
   virtual function void my_f1();
      my_f1_counter++;
   endfunction
endclass
```

```
initial begin
   groupExtension g1 = new ("my_group", "g1");
   groupExtension_callbacks cb1 = new();
   groupExtension_callbacks cb2 = new();
   g1.append_callback(cb1);
   g1.prepend_callback(cb2);
   ...
end
```

## vmm_group::unregister_callback()

Unregisters a callback.

### SystemVerilog

```
function void vmm_group::unregister_callback(
   vmm_group_callbacks cb);
```

### Description

Removes the specified callback extension `cb` to the callback registry, for this group.

### Example

```
class group_callbacks extends vmm_group_callbacks;
   virtual function void my_f1();
   endfunction
endclass
class groupExtension extends vmm_group;
   function new (string name, string inst,
         vmm_unit parent=null);
      super.new(name,inst,parent);
   endfunction
   function void build_ph();
      `vmm_callback(group_callbacks,my_f1());
   endfunction:build_ph
   ...
endclass
class groupExtension_callbacks extends group_callbacks;
   int my_f1_counter++;
   virtual function void my_f1();
      my_f1_counter++;
   endfunction
endclass
initial begin
groupExtension g1 = new ("my_group", "g1");
```

```
   groupExtension_callbacks cb1 = new();
   groupExtension_callbacks cb2 = new();
   g1.append_callback(cb1);
   g1.append_callback(cb2);
   ...
   g1.unregister_callback(cb2);
   ...
end
```

# vmm_log

The **vmm_log** class implements an interface to the message service.

Several methods apply to multiple message service interfaces, not just the one where the method is invoked. All message service interfaces that match the specified name and instance name are affected by these methods. If the name or instance name is enclosed between slashes (for example, “/.../”), then they are interpreted as *sed*-style regular expressions. If a value of "" is specified, then the name or instance name of the current message service interface is specified. If the **recurse** parameter is TRUE (non-zero), then all interfaces that are logically under the matching message service interfaces are also specified.

## Summary


- vmm_log::add_watchpoint() ..... page A-256
- vmm_log::append_callback() ..... page A-257
- vmm_log::catch() ..... page A-258
- vmm_log::copy() ..... page A-260
- vmm_log::create_watchpoint() ..... page A-261
- vmm_log::disable_types() ..... page A-262
- vmm_log::enable_types() ..... page A-264
- vmm_log::end_msg() ..... page A-266
- vmm_log::enum(message-severity) ..... page A-267
- vmm_log::enum(message-type) ..... page A-268
- vmm_log::enum(simulation-handling-value) ..... page A-270
- vmm_log::for_each() ..... page A-272
- vmm_log::get_instance() ..... page A-273
- vmm_log::get_message_count() ..... page A-274
- vmm_log::get_name() ..... page A-275
- vmm_log::get_verbosity() ..... page A-276
- vmm_log::is_above ..... page A-277
- vmm_log::kill() ..... page A-278
- vmm_log::list() ..... page A-279
- vmm_log::log_start() ..... page A-280
- vmm_log::log_stop() ..... page A-281
- vmm_log::modify() ..... page A-282
- vmm_log::new() ..... page A-283
- vmm_log::prepend_callback() ..... page A-284
- vmm_log::remove_watchpoint() ..... page A-285
- vmm_log::report() ..... page A-286

- vmm_log::reset() ..... page A-287
- vmm_log::set_instance() ..... page A-288
- vmm_log::set_name() ..... page A-289
- vmm_log::set_typ_image() ..... page A-290
- vmm_log::set_sev_image() ..... page A-291
- vmm_log::set_verbosity() ..... page A-293
- vmm_log::start_msg() ..... page A-294
- vmm_log::stop_after_n_errors() ..... page A-296
- vmm_log::text() ..... page A-297
- vmm_log::uncatch() ..... page A-299
- vmm_log::uncatch_all() ..... page A-300
- vmm_log::unmodify() ..... page A-301
- vmm_log::unregister_callback() ..... page A-302
- vmm_log::use_hier_inst_name() ..... page A-303
- vmm_log::use_orig_inst_name() ..... page A-305
- vmm_log::uses_hier_inst_name() ..... page A-306
- vmm_log::set_format() ..... page A-307
- vmm_log::wait_for_msg() ..... page A-308
- vmm_log::wait_for_watchpoint() ..... page A-309

## vmm_log::add_watchpoint()

Adds the specified watchpoint to the specified message service interfaces.

### SystemVerilog

```
virtual function void
           add_watchpoint(int watchpoint_id,
    string name = "",
    string inst = "",
    bit recurse = 0);
```

### OpenVera

Not supported.

### Description

Adds watchpoint as specified by `watchpoint_id` to the message interface specified by `name` and `inst` arguments. If a message matching the watchpoint specification is issued by one of the specified message service interfaces associated with the watchpoint, the watchpoint will be triggered. If the specified argument `recurse` is set, then this method also applies to all the message interfaces logically under the matching message service interfaces.

## vmm_log::append_callback()

Appends a callback façade instance with the message service.

### SystemVerilog

```
virtual function void
           append_callback(vmm_log_callbacks cb);
```

### OpenVera

Not supported.

### Description

Globally appends the specified callback façade instance with the message service. Callback methods are invoked in the order in which they were registered.

A warning is generated, if the same callback façade instance is registered more than once. Callback façade instances can be unregistered and re-registered dynamically.

### Example

*Example A-97*

```
class tb_env extends vmm_env;
   virtual function void build();
      ...
      begin
         sb_mac_cbs cb = new;
         this.mac.append_callback(cb);
      end
   endfunction: build
endclass: tb_env
```

## vmm_log::catch()

Adds a user-defined message handler.

### SystemVerilog

```
function int catch(
     vmm_log_catcher catcher,
     string name = "",
     string inst = "",
     bit recurse = 0,
     int typs = ALL_TYPS,
     int severity = ALL_SEVS,
     string text = "");
```

### OpenVera

Not supported.

### Description

Installs the specified message handler to catch any message of the specified type and severity, issued by the specified message service interface instances specified by `name` and `instance` arguments, which contains the specified text. By default, this method catches all messages issued by this message service interface instance. A unique message handler identifier is returned that can be used later to uninstall the message handler using the vmm_log::uncatch() method.

Messages are considered caught by the first found user-defined handler that can handle the message. User-defined message handlers are considered in reverse order of installation. This means that the last handler installed will be considered first. Once caught, messages are handed-off to the vmm_log_catcher::caught()

method, and will not be issued. A user-defined message handler may choose to explicitly issue the message using the vmm_log_catcher::issue() method, or throw the message back to the message service by using the vmm_log_catcher::throw() method, to be potentially caught by another suitable message handler or be issued.

Watchpoints are triggered after message catching. If the message is modified in the catcher, the modified message triggers applicable watchpoints, if any. If the specified argument `recurse` is set, then this method also applies to all the message interfaces logically under the matching message service interfaces.

## Example

*Example A-98*

```
class err_catcher extends vmm_log_catcher;
   ...
endclass

alu_env env;
err_catcher ctcher;

initial begin
  ...
  ctcher = new(10);
  ...
  env.build();
  env.sb.log.catch(ctcher,"","", ,vmm_log::ERROR_SEV,
      "/Mismatch/");
end
```

## vmm_log::copy()

Copies the configuration of this message service interface to the specified message service interface.

### SystemVerilog

```
virtual function vmm_log copy(vmm_log to = null);
```

### OpenVera

Not supported.

### Description

Copies the configuration of this message service interface to the specified message service interface (or a new interface, if none is specified), and returns a reference to the interface copy. The current configuration of the message service interface is copied, except the hierarchical relationship information, which is not modified.

## vmm_log::create_watchpoint()

Creates a watchpoint descriptor.

### SystemVerilog

```
virtual function int
           create_watchpoint(int types = ALL_TYPS,
     int severity = ALL_SEVS,
     string text = "",
     logic issued = 1'bx);
```

### OpenVera

Not supported.

### Description

Creates a watchpoint descriptor that will be triggered when the specified message is used. The message can be specified by type, severity, or by text pattern. By default, messages of all types, severities, and text are specified. A message must match all specified criteria to trigger the watchpoint. The **`issued`** parameter specifies if the watchpoint is triggered when the message is physically issued (1'b1), physically not issued (filtered out (1'b0)), or regardless if the message is physically issued or not (1'bx).

A watchpoint will be repeatedly triggered, every time a message matching the watchpoint specification is generated by a message service interface associated with the watchpoint.

## vmm_log::disable_types()

Specifies the message types to be disabled by the specified message service interfaces.

### SystemVerilog

```
virtual function void disable_types(int typs,
     string name = "",
     string inst = "",
     bit recursive = 0);
```

### OpenVera

Not supported.

### Description

Specifies the message types to be disabled by the specified message service interfaces. Message service interfaces are specified by a value or regular expression, for both the name and instance name. If no name and no instance are explicitly specified, then this message service interface is implicitly specified.

If the name or instance named are specified between “/” characters, then the specification is interpreted as a regular expression that must be matched against all known names and instance names, respectively. Both names must match to consider a message service interface as specified. If the **`recursive`** argument is TRUE, then all message service interfaces that are hierarchically below the specified message service interfaces, are included in the specification, whether their name and instance name matches or not. A message service interface must exist to be specified.

The **`types`** argument specifies the message types to enable or disable. Types are specified as the bitwise-or or sum of all relevant types.

By default, all message types are enabled.

## vmm_log::enable_types()

Specifies the message types to be displayed by the specified message service interfaces.

### SystemVerilog

```
virtual function void enable_types(int typs,
     string name = "",
     string inst = "",
     bit recursive = 0);
```

### OpenVera

Not supported.

### Description

Specifies the message types to be displayed by the specified message service interfaces. Message service interfaces are specified by a value or regular expression for both the name and instance name. If no name and no instance are explicitly specified, then this message service interface is implicitly specified.

If the name or instance named are specified between “/” characters, then the specification is interpreted as a regular expression that must be matched against all known names and instance names, respectively. Both names must match to consider a message service interface, as specified. If the **`recursive`** argument is TRUE, all message service interfaces that are hierarchically below the specified message service interfaces are included in the specification, whether their name and instance name matches or not. A message service interface to be specified, must exist.

The **`types`** argument specifies the message types to enable or disable. Types are specified as the bitwise-or or sum of all relevant types.

By default, all message types are enabled.

## vmm_log::end_msg()

Flushes and terminates the current message.

### SystemVerilog

```
virtual function void end_msg();
```

### OpenVera

Not supported.

### Description

Flushes and terminates the current message, and triggers the message display and the simulation handling. A message can be flushed multiple times using the **`vmm_log::text("")`** method, but the simulation handling and notification will only take effect on message termination.

## vmm_log::enum(*message-severity*)

Enumerated type defining symbolic values for message severities.

### SystemVerilog

```
enum int {FATAL_SEV   = 'h0001,
          ERROR_SEV   = 'h0002,
          WARNING_SEV = 'h0004,
          NORMAL_SEV  = 'h0008,
          TRACE_SEV   = 'h0010,
          DEBUG_SEV   = 'h0020,
          VERBOSE_SEV = 'h0040,
          HIDDEN_SEV  = 'h0080,
          IGNORE_SEV  = 'h0100,
          DEFAULT_SEV = -1,
          ALL_SEVS    = 'hFFFF
          } severities_e;
```

### OpenVera

Not supported.

### Description

Enumerated type defining symbolic values for message severities used, when specifying a message severity in properties or method arguments. The **`vmm_log::DEFAULT_SEV`** and **`vmm_log::ALL_SEVS`** are special symbolic values usable only with some control methods, and are not used to issue actual messages. Multiple message severities can be specified to some control methods by combining the value of the required severities using the bitwise-or or addition operator.

## vmm_log::enum(*message-type*)

Enumerated type defining symbolic values for message types.

### SystemVerilog

```
enum int {FAILURE_TYP     = 'h0001,
          NOTE_TYP        = 'h0002,
          DEBUG_TYP       = 'h0004,
          REPORT_TYP      = 'h0008,
          NOTIFY_TYP      = 'h0010,
          TIMING_TYP      = 'h0020,
          XHANDLING_TYP   = 'h0040,
          PROTOCOL_TYP    = 'h0080,
          TRANSACTION_TYP = 'h0100,
          COMMAND_TYP     = 'h0200,
          CYCLE_TYP       = 'h0400,
          USER_TYP_0      = 'h0800,
          USER_TYP_1      = 'h1000,
          USER_TYP_2      = 'h2000,
          INTERNAL_TYP    = 'h4000,
          DEFAULT_TYP     = -1,
          ALL_TYPS        = 'hFFFF
          } types_e;
```

### OpenVera

Not supported.

### Description

Enumerated type defining symbolic values for message types used, when specifying a message type in properties or method arguments. The **`vmm_log::DEFAULT_TYP`** and **`vmm_log::ALL_TYPS`** are special symbolic values usable only with some control methods, and

are not used to issue actual messages. Multiple message types can be specified to some control methods by combining the value of the required types, using the bitwise-or or addition operator.

## vmm_log::enum(*simulation-handling-value*)

Symbolic values for simulation handling.

### SystemVerilog

```
enum int {CONTINUE         = 'h0001,
          COUNT_ERROR      = 'h0002,
          DEBUGGER         = 'h0004,
          DUMP_STACK       = 'h0008,
          STOP_PROMPT      = 'h0010,
          ABORT_SIM        = 'h0020,
          IGNORE           = 'h0040,
          DEFAULT_HANDLING = -1
          } handling_e;
```

### OpenVera

Not supported.

### Description

Enumerated type defining symbolic values for simulation handling used, when specifying a new simulation handling when promoting or demoting a message using the **`vmm_log::modify()`** method.

Unless this method is specified, message types are assigned the default severity and simulation handling, as shown in Table A-8.

*Table A-8 Default Message Severities and Handling*

*Table A-9*

| **Message Type** | **Default Severity** | **Default Handling** |
|---|---|---|
| FAILURE_TYP | ERROR_SEV | COUNT_ERROR |
| NOTE_TYP | NORMAL_SEV | CONTINUE |
| DEBUG_TYP | DEBUG_SEV | CONTINUE |
| TIMING_TYP XHANDLING_TYP | WARNING_SEV | CONTINUE |
| TRANSACTION_TYP COMMAND_TYP | TRACE_SEV | CONTINUE |
| REPORT_TYP PROTOCOL_TYP | DEBUG_SEV | CONTINUE |
| CYCLE_TYP | VERBOSE_SEV | CONTINUE |
| Any type | FATAL_SEV | ABORT_SIM |

## vmm_log::for_each()

Iterates over message service instances.

### SystemVerilog

```
function vmm_log for_each();
```

### OpenVera

```
function rvm_log for_each();
```

### Description

Returns a reference to the next known message service interface that matches the iterator specification, specified in the last invocation of "vmm_log::reset()" method. Returns *NULL,* if no more instances match.

There is one iterator per message service instance.

### Example

*Example A-99*

```
env.log.reset();
for (vmm_log log = env.log.for_each();
     log != null;
     log = env.log.for_each()) begin
   ...
end
```

## vmm_log::get_instance()

Returns the instance name of the message service interface.

### SystemVerilog

```
virtual function string get_instance();
```

### OpenVera

Not supported.

### Description

This method returns the instance name of the message service interface.

## vmm_log::get_message_count()

Returns the total number of messages of the specified severities.

### SystemVerilog

```
virtual function int
           get_message_count(int severity = ALL_SEVS,
    string name = "",
    string instance = "",
    bit recurse = 0);
```

### OpenVera

Not supported.

### Description

Returns the total number of messages of the specified severities that are issued from the specified message service interfaces. Message severities can be specified as a sum of individual message severities to specify more than one severity.

## vmm_log::get_name()

Returns the nam of the message service interface.

### SystemVerilog

```
virtual function string get_name();
```

### OpenVera

Not supported.

### Description

This method returns the name of the message service interface.

## vmm_log::get_verbosity()

Returns the minimum message severity to be displayed.

### SystemVerilog

```
virtual function int get_verbosity();
```

### OpenVera

Not supported.

### Description

Returns the minimum message severity to be displayed, when sourced by this message service interface.

## vmm_log::is_above

Specifies that this message service instance is hierarchically above the specified message service interface.

### SystemVerilog

```
virtual function void is_above(vmm_log log);
```

### OpenVera

Not supported.

### Description

This method is the corollary of the **`under`** argument of the constructor, and need not be used if the specified message service interface has already been constructed as being under this message service interface.

## vmm_log::kill()

Removes internal references to the message service interface.

### SystemVerilog

```
virtual function void kill();
```

### OpenVera

Not supported.

### Description

Removes any internal reference to this message service interface, so that it may be reclaimed by the garbage collection, once all use references are also removed. Once this method is invoked, it is no longer possible to control this message service interface by name.

## vmm_log::list()

Lists message service interfaces that match a specified name and instance name.

### SystemVerilog

```
virtual function void list(string name = "/./",
     string instance = "/./",
     bit recurse = 0);
```

### OpenVera

Not supported.

### Description

Lists all message service interfaces that match the specified name and instance name. If the **`recurse`** parameter is TRUE (non-zero), then all interfaces that are logically under the matching message service interface are also listed.

## vmm_log::log_start()

Appends all messages produced by the specified message service interfaces.

### SystemVerilog

```
virtual function void log_start(int file,
     string name = "",
     string instance = "",
     bit recurse = 0)
```

### OpenVera

Not supported.

### Description

Appends all messages produced by the specified message service interfaces to the specified file. The **`file`** argument must be a file descriptor, as returned by the **`$fopen()`** system task. By default, all message service interfaces append their messages to the standard output. Specifying a new output file does not stop messages from being appended to previously specified files.

## vmm_log::log_stop()

Stops logging messages from a specified message service interface.

### SystemVerilog

```
virtual function void log_stop(int file,
    string name = "",
    string instance = "",
    bit recurse = 0);
```

### OpenVera

Not supported.

### Description

Messages issued by the specified message service interfaces are no longer appended to the specified file. The **`file`** argument must be a file descriptor, as returned by the **`$fopen()`** system task. If the specified **`file`** argument is 0, then messages are no longer sent to the standard simulation output and transcript. If the **`file`** argument is specified as –1, then appending to all files, except the standard output, is stopped.

## vmm_log::modify()

Modifies the specified type, severity, or simulation handling for a message source.

### SystemVerilog

```
virtual function int
   modify(string name = "",
     string inst = "",
     bit recursive = 0,
     int typ = ALL_TYPS,
     int severity = ALL_SEVS,
     string text = "",
     int new_typ = UNCHANGED,
     int new_severity = UNCHANGED,
     int handling = UNCHANGED);
```

### OpenVera

Not supported.

### Description

Modifies the specified message source by any of the specified message service interfaces, with the new specified type, severity, or simulation handling. The message can be specified by type, severity, numeric ID, or by text pattern. By default, messages of any type, severity, ID, or text is specified. A message must match all specified criteria.

This method returns a unique message modifier identifier that can be used to remove it using the **`vmm_log::unmodify()`** method. All message modifiers are applied in the same order they were defined, before a message is generated.

## vmm_log::new()

Creates a new instance of a message service interface.

### SystemVerilog

```
function new(string name,
     string inst,
     vmm_log under = null);
```

### OpenVera

Not supported.

### Description

Creates a new instance of a message service interface, with the specified interface name and instance name. Moreover, a message service interface can optionally be specified as hierarchically below another message service instance, to create a logical hierarchy of message service interfaces.

## vmm_log::prepend_callback()

Prepends a callback façade instance with the message service.

### SystemVerilog

```
virtual function void
           prepend_callback(vmm_log_callbacks cb);
```

### OpenVera

Not supported.

### Description

Globally prepends the specified callback façade instance with the message service. Callback methods will be invoked in the order in which they were registered.

A warning is generated if the same callback façade instance is registered more than once. Callback façade instances can be unregistered and re-registered dynamically.

### Example

*Example A-100*

```
env.build();
begin
   gen_rx_errs cb = new;
   env.phy.prepend_callback(cb);
end
```

## vmm_log::remove_watchpoint()

Removes the specified watchpoint from the specified message service interfaces.

### SystemVerilog

```
virtual function void remove_watchpoint(
                            int watchpoint_id=-1,
                            string name = "",
                            string inst = "",
                            bit recurse = 0);
```

### OpenVera

Not supported.

### Description

Removes the specified watchpoint `watchpoint_id` from the message interface specified by `name` and `instance` arguments. If a message matching the watchpoint specification is issued by one of the specified message service interfaces associated with the watchpoint, the watchpoint will be triggered. If the specified argument `recurse` is set, then this method also applies to all the message interfaces logically under the matching message service interfaces.

## vmm_log::report()

Reports a failure, if a message service interface issued an error or fatal message.

### SystemVerilog

```
virtual task report(   string name = "/./",
                       string inst = "/./",
                       bit recurse = 0);
```

### OpenVera

Not supported.

### Description

Reports a failure if any of the specified message service interfaces, matched by `name` and `inst` arguments, have issued any error or fatal messages. Reports a success otherwise. The text of the pass or fail message is specified using the `vmm_log_format::pass_or_fail()` method. If the specified argument `recurse` is set, then this method also applies to all the message interfaces logically under the matching message service interfaces.

## vmm_log::reset()

Initializes the message service instance iterator.

### SystemVerilog

```
function void reset(string name    = "/./",
     string inst    = "/./",
     bit    recurse = 0);
```

### OpenVera

```
task reset(string name    = "/./",
     string inst    = "/./",
     bit    recurse = 0);
```

### Description

Resets the message service instance iterator for this instance of the message service, and initialize it to iterator using the specified name, instance name, and optional recursion.

It is then possible to iterate over all known instances of the message service interface that match the specified pattern, using the "vmm_log::for_each()" method.

There is one iterator per message service instance.

### Example

*Example A-101*

```
env.log.reset();
for (vmm_log log = env.log.for_each();
     log != null;
     log = env.log.for_each()) begin
end
```

## vmm_log::set_instance()

Sets the instance name of the message service interface.

### SystemVerilog

```
virtual function void set_instance(string inst);
```

### OpenVera

Not supported.

### Description

This method sets the instance name of the message service interface.

## vmm_log::set_name()

Sets the name of the message service interface.

### SystemVerilog

```
virtual function void set_name(string name);
```

### OpenVera

Not supported.

### Description

This method sets the name of the message service interface.

## vmm_log::set_typ_image()

Replaces the image, which is used to display the specified message.

### SystemVerilog

```
virtual function string set_typ_image(int typ,
          string image);
```

### OpenVera

Not supported.

### Description

Globally replaces the image, which is used to display the specified message type with the specified image. The previous image is returned. Default images are provided.

Default colors for fatal, error, and warning messages can be automatically selected by using `+define+VMM_LOG_ANSI_COLOR`.

Messages can be custom color coded by specifying the ANSI escape characters with the `set_sev_image()` or `set_typ_image()` methods.

For example,

```
log.set_sev_image( vmm_log::FATAL_SEV,
"\033[41m*FATAL*\033[0m");
```

## vmm_log::set_sev_image()

Replaces the image, which is used to display the specified message severity.

### SystemVerilog

```
virtual function string set_sev_image(int severity,
            string image);
```

### OpenVera

Not supported.

### Description

Globally replaces the image, which is used to display the specified message severity with the specified image. The previous image is returned. Default images are provided.

Default colors for fatal, error, and warning messages can be automatically selected by using `+define+VMM_LOG_ANSI_COLOR`.

Messages can be custom color coded by specifying the ANSI escape characters with the `set_sev_image()` or `set_typ_image()` methods.

For example,

```
log.set_sev_image( vmm_log::FATAL_SEV,
"\033[41m*FATAL*\033[0m");
```

## Example

*Example A-102*

Following is an example for colorizing the severity display on ANSI terminals.

```
log.set_sev_image(vmm_log::WARNING,
                  "\033[33mWARNING\033[0m");
log.set_sev_image(vmm_log::ERROR_SEV,
                  "\033[31mERROR\033[0m");
log.set_sev_image(vmm_log::FATAL_SEV,
                   "\033[41m*FATAL*\033[0m");
```

## vmm_log::set_verbosity()

Specifies the minimum message severity to be displayed.

### SystemVerilog

```
virtual function void set_verbosity(int severity,
     string name = "",
     string inst = "",
     bit recursive = 0);
```

### OpenVera

Not supported.

### Description

Specifies the minimum message severity to be displayed, when sourced by the specified message service interfaces. For ore information, see the documentation for the **`enable_types()`** method for the interpretation of the **`name`**, **`inst,`** and **`recursive`** arguments, and how they are used to specify message service interfaces.

The default minimum severity can be changed by using the **`+vmm_log_default=`***`sev`* runtime command-line option, where *`sev`* is the desired minimum severity and is one of the levels such as **`error`**, **`warning`**, **`normal`**, **`trace`**, **`debug,`** or **`verbose`**. The default verbosity level can be later modified using this method.

The minimum severity level can be globally forced by using the **`+vmm_force_verbosity=`***`sev`* runtime command-line option. The specified verbosity overrides the verbosity level specified, using this method.

## vmm_log::start_msg()

Prepares to generate a message.

### SystemVerilog

```
virtual function bit start_msg( int typ, int severity =
DEFAULT_SEV);

With +define VMM_LOG_FORMAT_FILE_LINE
virtual function bit start_msg( int typ,
                                int severity = DEFAULT_SEV,
                                string fname = "",
                                int     line  = -1);
```

### OpenVera

```
virtual function bit (integer type,
    integer severity = DEFAULT_TYP,
    integer msg_id = -1);
```

### Description

Prepares to generate a message of the specified type and severity. If the message service interface instance is configured ignore messages of the specified type or severity, then the function returns `FALSE`.

When using SystemVerilog, the current filename and line number, using `fname` and `line` arguments, where the message is created can be supplied by using the `` `__FILE__ `` and `` `__LINE__ `` symbols. For backward compatibility, the `` `VMM_LOG_FORMAT_FILE_LINE `` symbol must be defined to enable the inclusion of the filename, and line number to the message formatter.

## Example

*Example A-103*

```
program test
   ...
   initial begin
      ...
      env.log.text.start_msg(vmm_log::NOTE_TYP,
                             vmm_log::DEFAULT_SEV,
                             `__FILE__,`__LINE__);
      env.log.text("Starting Test My_Test");
      env.log.text();
      ...
   end
```

## vmm_log::stop_after_n_errors()

Aborts the simulation, after a specified number of messages are generated.

### SystemVerilog

```
virtual function void stop_after_n_errors(int n);
```

### OpenVera

Not supported.

### Description

Aborts the simulation, after the specified number of messages with a simulation handling of **`COUNT_ERROR`** is generated. This value is global, and all messages from any message service interface count toward this limit. A zero or negative value specifies no maximum. The default value is 10. The message specified by the **`vmm_log_format::abort_on_error()`** is displayed, before the simulation is aborted.

## vmm_log::text()

Adds the specified text to the message being constructed.

### SystemVerilog

```
virtual function bit text(string msg = "");
```

### OpenVera

Not supported.

### Description

Adds the specified text to the message being constructed. This method specifies a single line of message text. A newline character is automatically appended when the message is issued. Additional lines of messages can be produced by calling this method multiple times, once per line. If an empty string is specified as message text, all previously specified lines of text are flushed to the output, but the message is not terminated. This method may return FALSE, if the message is filtered out based on the text.

A message must be flushed and terminated by calling the **`vmm_log::end_msg()`** method, to trigger the message display and the simulation handling. A message can be flushed multiple times by calling the **`vmm_log::text("")`** method, but the simulation handling and notification will take effect on the message termination.

If additional lines are produced using the **`$display()`** system task or other display mechanisms, they will not be considered by the filters, nor included in explicit log files. They may also be displayed out of order, if they are produced before the previous lines of the message are flushed.

For single-line messages, the **`'vmm_fatal()`**, **`'vmm_error()`**, **`'vmm_warning()`**, **`'vmm_note()`**, **`'vmm_trace()`**, **`'vmm_debug()`**, **`'vmm_verbose()`**, **`'vmm_report()`**, **`'vmm_command()`**, **`'vmm_transaction()`**, **`'vmm_protocol(),`** and **`'vmm_cycle()`** macros can be used as a shorthand notation.

*Table A-10 Message Type and Severity for Shorthand Macros*

*Table A-11*

| **Macro** | **Message Type** | **Message Severity** |
|---|---|---|
| ‘vmm_fatal(vmm_log log,  string txt); | Failure | Fatal |
| ‘vmm_error(vmm_log log, string txt); | Failure | Error |
| ‘vmm_warning(vmm_log log, string txt); | Failure | Warning |
| ‘vmm_note(vmm_log log, string txt); | Note | Default |
| ‘vmm_trace(vmm_log log, string txt); | Debug | Trace |
| ‘vmm_debug(vmm_log log, string txt); | Debug | Debug |
| ‘vmm_verbose(vmm_log log, string txt); | Debug | Verbose |
| ‘vmm_report(vmm_log log, string txt); | Report | Default |
| ‘vmm_command(vmm_log log, string txt); | Command | Default |
| ‘vmm_transaction(vmm_log log, string txt); | Transaction | Default |
| ‘vmm_protocol(vmm_log log, string txt); | Protocol | Default |
| ‘vmm_cycle(vmm_log log, string txt); | Cycle | Default |

## vmm_log::uncatch()

Removes a user-defined message handler.

### SystemVerilog

```
function bit uncatch(int catcher_id);
```

### OpenVera

Not supported.

### Description

Uninstalls the specified user-defined message handler. The message handler is identified by the unique identifier that was returned by the `vmm_log::catch()` method, when it was originally installed.

Returns `TRUE`, if the specified message handler was successfully uninstalled. Otherwise, it returns `FALSE`.

### Example

*Example A-104*

```
class err_catcher extends vmm_log_catcher;
endclass
alu_env env;
err_catcher ctcher;
initial begin
  env.build();
  ctcher_id = env.sb.log.catch(ctcher, , , ,
      vmm_log::ERROR_SEV,"/Mismatch/");
  env.sb.log.uncatch(ctcher_id);
end
```

## vmm_log::uncatch_all()

Removes all user-defined message handlers.

### SystemVerilog

```
function void uncatch_all();
```

### OpenVera

Not supported.

### Description

Uninstalls all user-defined message handlers. All message handlers, even those that were registered with or through a different message service interface, are uninstalled.

### Example

*Example A-105*

```
class err_catcher extends vmm_log_catcher;
endclass

alu_env env;
err_catcher ctcher1, ctcher2;

initial begin
  env.build();
  ctcher_id1 = env.log.catch(ctcher1, , , ,
     vmm_log::ERROR_SEV,"/MON_ERROR_008/");
  ctcher_id2 = env.log.catch(ctcher2, , , ,
      vmm_log::ERROR_SEV,"/MON_ERROR_010/");
  if(env.mon.error_cnt >10)
      env.log.uncatch_all();
end
```

## vmm_log::unmodify()

Removes a message modification from the message service interfaces.

### SystemVerilog

```
virtual function void unmodify(int modification_id = -1,
     string name = "",
     string instance = "",
     bit recursive = 0);
```

### OpenVera

Not supported.

### Description

Removes the specified message `modification_id` from the specified message service interfaces. By default, all message modifications are removed. If the specified argument `recursive` is set, then this method also applies to all the message interfaces logically under the matching message service interfaces.

## vmm_log::unregister_callback()

Unregisters the specified callback façade instance.

### SystemVerilog

```
virtual function void unregister_callback(
    vmm_log_callbacks cb);
```

### OpenVera

Not supported.

### Description

Globally unregisters the specified callback façade instance with the message service. A warning is generated, if the specified façade instance is not currently registered with the service. Callback façade instances can later be re-registered.

## vmm_log::use_hier_inst_name()

Switches to hierarchical instance names.

### SystemVerilog

```
function void use_hier_inst_name();
```

### OpenVera

Not supported.

### Description

Rewrites the instance name of all message service interface instances into a dot-separated hierarchical form. The original instance names can later be restored using the "vmm_log::use_orig_inst_name()" method.

An instance name is made hierarchical, if the message service instance is specified as being under another message service interface. Message service interface hierarchies can be built by specifying the *under* argument to the constructor, or by using the vmm_log::is_above() method.

For example, the code in Example A-106 results in instance names such as top, top.m1, top.c1, and s1. The instance name for s1 is not modified, because it is not specified as being under another message service interface, and thus creates a new hierarchical root.

## Example

*Example A-106*

```
function tb_env::new();
   super.new("top");
endfunction

function void tb_env::build();
   super.build();
   this.chan = new("Master to slave", "c1");
   this.master = new("m1", this.chan);
   this.slave  = new("s1", this.chan);
   this.log.is_above(this.master.log);
   this.log.is_above(this.chan);
   this.log.use_hier_inst_name();
endfunction
```

## vmm_log::use_orig_inst_name()

Switches to original, flat instance names.

### SystemVerilog

```
function void use_orig_inst_name();
```

### OpenVera

Not supported.

### Description

Rewrites the instance name of all message service interface instances into the original and flat form specified, when the message service instance was constructed.

### Example

*Example A-107*

```
env.build();
if (env.log.uses_hier_inst_name())
   env.log.use_orig_inst_name();
```

## vmm_log::uses_hier_inst_name()

Checks if hierarchical instance names are in use.

### SystemVerilog

```
function bit uses_hier_inst_name();
```

### OpenVera

Not supported.

### Description

Returns `TRUE`, if the message service interface instances use hierarchical instance name, as defined by calling the "vmm_log::use_hier_inst_name()" method. Returns `FALSE`, if the original and flat instance names are used, as defined by calling the "vmm_log::use_orig_inst_name()" method.

### Example

*Example A-108*

```
env.build();
if (!env.log.uses_hier_inst_name())
   env.log.use_hier_inst_name();
```

## vmm_log::set_format()

Sets the message formatter to the specified message formatter instance.

### SystemVerilog

```
virtual function vmm_log_format
     set_format(vmm_log_format fmt);
```

### OpenVera

Not supported.

### Description

Globally sets the message formatter to the specified message formatter instance. A reference to the previously used message formatter instance is returned. A default global message formatter is provided.

## vmm_log::wait_for_msg()

Waits for a one-time watchpoint for a specified message.

### SystemVerilog

```
virtual task wait_for_msg(string name = "",
     string inst = "",
     bit recurse = 0,
          int typs = ALL_TYPS,
     int severity = ALL_SEVS,
     string text = "",
     logic issued = 1'bx,
     ref vmm_log_msg msg);
```

### OpenVera

Not supported.

### Description

Sets up and waits for a one-time watchpoint for the specified message (described by `severity`, message type `typs`, and string text) on the specified message service interface (specified by `inst` and `name`). The watchpoint is triggered only once and removed after being triggered. If the specified argument `recurse` is set, then this method also applies to all the message interfaces logically under the matching message service interfaces.

A descriptor of the message that triggered the watchpoint will be updated to the reference argument `msg`. Argument `issued` keeps track whether the message is issued or not

## vmm_log::wait_for_watchpoint()

Waits for the specified watchpoint to be triggered by a message.

### SystemVerilog

```
virtual task wait_for_watchpoint(int watchpoint_id,
     ref vmm_log_msg msg);
```

### OpenVera

Not supported.

### Description

Waits for the specified watchpoint to be triggered by a message issued by one of the message service interfaces attached to the watchpoint. A descriptor of the message that triggered the watchpoint will be updated to the reference argument `msg`.

# vmm_log_msg

This class describes a message issued by a message service interface that caused a watchpoint to be triggered. It is returned by the **`vmm_log::wait_for_watchpoint()`** and **`vmm_log::wait_for_msg()`** method.

## Summary


- vmm_log_msg::effective_severity ..... page A-311
- vmm_log_msg::effective_typ ..... page A-312
- vmm_log_msg::handling ..... page A-313
- vmm_log_msg::issued ..... page A-314
- vmm_log_msg::log ..... page A-315
- vmm_log_msg::original_severity ..... page A-316
- vmm_log_msg::original_typ ..... page A-317
- vmm_log_msg::text[] ..... page A-318
- vmm_log_msg::timestamp ..... page A-319

## vmm_log_msg::effective_severity

Effective message severity as potentially modified by the **`vmm_log::modify()`** method.

### SystemVerilog

```
int effective_severity;
```

### OpenVera

Not supported.

## vmm_log_msg::effective_typ

Effective message type as potentially modified by the **`vmm_log::modify()`** method.

### SystemVerilog

```
int effective_typ;
```

### OpenVera

Not supported.

## vmm_log_msg::handling

The simulation handling after the message is physically generated.

### SystemVerilog

```
int handling;
```

### OpenVera

Not supported.

## vmm_log_msg::issued

Indicates if the message is physically generated or not.

### SystemVerilog

```
logic issued;
```

### OpenVera

Not supported.

### Description

If non-zero, then the message is generated.

## vmm_log_msg::log

A reference to the message reporting interface that has generated the message.

### SystemVerilog

```
vmm_log log;
```

### OpenVera

Not supported.

## vmm_log_msg::original_severity

Original message severity, as specified in the code creating the message.

### SystemVerilog

```
int original_severity;
```

### OpenVera

Not supported.

## vmm_log_msg::original_typ

Original message type, as specified in the code creating the message.

### SystemVerilog

```
int original_typ;
```

### OpenVera

Not supported.

## vmm_log_msg::text[]

Formatted text of the message.

### SystemVerilog

```
string text[$];
```

### OpenVera

Not supported.

### Description

Each element of the array contains one line of text, as built by individual calls to the **`vmm_log::text()`** method.

## vmm_log_msg::timestamp

The simulation time when the message was generated.

### SystemVerilog

```
time timestamp;
```

### OpenVera

Not supported.

# vmm_log_callback

This class provides a facade for the callback methods provided by the message service. Callbacks are associated with the message service itself, but not a particular message service interface instance.

## Summary


- vmm_log_callback::pre_abort() ..... page A-321
- vmm_log_callback::pre_debug() ..... page A-322
- vmm_log_callback::pre_finish() ..... page A-323
- vmm_log_callback::pre_stop() ..... page A-325

## vmm_log_callback::pre_abort()

Aborts the condition callback.

### SystemVerilog

```
virtual function void pre_abort(vmm_log log);
```

### OpenVera

```
virtual function pre_abort(rvm_log log);
```

### Description

This callback method is invoked by the message service when a message was generated with an `ABORT` simulation handling, or the maximum number of message with a `COUNT_ERROR` handling is generated. This callback method is invoked before the “vmm_log_callback::pre_finish()” callback method.

The message service interface provided as an argument may be used to generate further messages.

## vmm_log_callback::pre_debug()

Debugs condition callback.

### SystemVerilog

```
virtual function void pre_debug(vmm_log log);
```

### OpenVera

```
virtual function pre_debug(rvm_log log);
```

### Description

This callback method is invoked by the message service when a message was generated with a DEBUGGER simulation handling.

The message service interface provided as an argument may be used to generate further messages.

## vmm_log_callback::pre_finish()

Terminates the simulation callback.

### SystemVerilog

```
virtual function void pre_finish(vmm_log log,
                                 ref bit finished);
```

### OpenVera

Not supported.

### Description

This callback method is invoked by the message service, after the "vmm_log_callback::pre_abort()" callback method, and immediately before the `$finish()` method is invoked to terminate the simulation.

The value of the `finished` parameter is 0 by default. If its value is returned as 1, by the sequence of callback methods, it indicates that the callback method is taken the responsibility of terminating the simulation. Therefore, the final report and the `$finish()` method will not be called.

Use this callback method, if you wish to delay the termination of the simulation, after an abort condition is detected.

### Example

*Example A-109 Terminating the Simulation of 100 Time Units*

```
virtual function void pre_finish(vmm_log log,
                                 ref bit finished);
```

```
      fork
         begin
            #100;
            log.report();
            $finish();
         end
      join_none
      finished = 1;
   endfunction
```

## vmm_log_callback::pre_stop()

Stops the condition callback.

### SystemVerilog

```
virtual function void pre_stop(vmm_log log);
```

### OpenVera

```
virtual function pre_stop(rvm_log log);
```

### Description

This callback method is invoked by the message service when a message was issued with a STOP simulation handling.

The message service interface provided as an argument may be used to issue further messages.

# vmm_log_catcher

VMM provides a mechanism to execute user-specific code, if a certain message is generated from the testbench environment, using the **vmm_log_catcher** class.

The **vmm_log_catcher** class is based on regexp to specify matching **vmm_log** messages.

If a message with the specified regexp is generated during simulation, the user-specified code is executed.

The **vmm_log_catcher::caught()** method can be used to modify the caught message, change its type and severity. You can choose to ignore the message, in which case it is not displayed. The message can be displayed as is, after executing user-specified code. The updated message can be displayed by calling the **vmm_log_catcher::issue()** method, in the caught method.

The caught message, modified or unmodified, can be passed to other catchers that are registered, using the **vmm_log_catcher::throw** function.

The messages to be caught are registered with the **vmm_log** class using the **vmm_log::catch()** method.

```
class error_catcher extends vmm_log_catcher;

virtual function void caught(vmm_log_msg msg);
   msg.text[0] = {" Acceptable Error" , msg.text[0]};
   msg.effective_severity = vmm_log::WARNING_SEV;
   issue(msg);
endfunction
endclass
```

Registration should be done in the program block.

```
initial begin
   env = new();
   error_catcher catcher = new();
   env.build();
   catcher_id =
      env.sb.log.catch(catcher,,,1,,vmm_log::ERROR_SEV,"/
      Mismatch/");
   env.run();
end
```

The **`error_catcher`** class extends the **`vmm_log_catcher`** class and implements the **`caught()`** method. The **`caught()`** method prepends the "Acceptable Error" to the original message, and changes the severity to `WARNING_SEV`.

In the initial block of the program block, an object of **`error_catcher`** is created, and a handle passed to the **`catch()`** method to register the catcher. Any **`vmm_log`** message from scoreboard (sb), containing the `ERROR_SEV` severity, and including the string "Mismatch" is caught and changed to `WARNING_SEV` with "Acceptable Error" prepended to it.

If the message is to be caught from all **`vmm_log`** instances, the **`catch()`** method can be called as:

```
env.sb.log.catch(catcher,"/./","/./
",1,vmm_log::ERROR_SEV,"/Mismatch/");
```

To unregister a catcher, the **`vmm_log::uncatch(`***`catcher-id`***`)`** or **`vmm_log::uncatch_all()`** methods can be used.

## Summary


- vmm_log_catcher::caught() ..... page A-329

- vmm_log_catcher::issue() ..... page A-331
- vmm_log_catcher::throw() ..... page A-332

## vmm_log_catcher::caught()

Handles a caught message.

### SystemVerilog

```
virtual function void caught(vmm_log_msg msg);
```

### OpenVera

Not supported.

### Description

This method specifies how to handle a caught message. Unless re-generated using the `vmm_log_catcher::issue()` method, or thrown back to the message service using the `vmm_log_catcher::throw()` method, this message will not be generated.

It is up to you to decide how a message, once caught, is to be handled. Handling a message is defined by whatever behavior is specified in the extension of this method. If left empty, the message will be ignored.

This method must be overloaded.

### Example

*Example A-110*

```
virtual function void caught(vmm_log_msg msg);
   if (num_errors < max_errors) begin
      msg.text[0] = {"ACCEPTABLE ERROR: ", msg.text[0]};
      msg.effective_severity = vmm_log::WARNING_SEV;
```

```
        ...
    end
  else
        . .
endfunction
```

## vmm_log_catcher::issue()

Generates a caught message.

### SystemVerilog

```
protected function void issue(vmm_log_msg msg);
```

### OpenVera

Not supported.

### Description

Immediately generates the specified message. The message is not subject to being caught any further by this or another user-defined message handler.

The message described by the `vmm_log_msg` descriptor may be modified before being generated.

### Example

*Example A-111*

```
virtual function void caught(vmm_log_msg msg);
   if (num_errors > max_errors) begin
      issue(msg);
   end
   ...
endfunction
```

## vmm_log_catcher::throw()

Throws back a caught message.

### SystemVerilog

```
protected function void throw(vmm_log_msg msg);
```

### OpenVera

Not supported.

### Description

Throws the specified message back to the message service. The message will be subject to being caught another user-defined message handler, but not by this one.

The message described by the `vmm_log_msg` descriptor may be modified before being thrown back.

### Example

*Example A-112*

```
virtual function void caught(vmm_log_msg msg);
   if (num_errors < max_errors)
   throw(msg);
endfunction
```

# vmm_log_format

This class is used to specify how messages are formatted, before being displayed or logged to files. The default implementation of these methods produces the default message format.

## Summary


- vmm_log_format::abort_on_error() ..... page A-334
- vmm_log_format::continue_msg() ..... page A-335
- vmm_log_format::format_msg() ..... page A-337
- vmm_log_format::pass_or_fail() ..... page A-339

## vmm_log_format::abort_on_error()

Called when the total number of **`COUNT_ERROR`** messages exceeds the error message threshold.

### SystemVerilog

```
virtual function string abort_on_error(int count,
     int limit);
```

### OpenVera

Not supported.

### Description

The string returned by the method describes the cause of the simulation aborting. If *`null`* is returned, then no explanation is displayed.

This method is called and the returned string is displayed, before the **`vmm_log_callbacks::pre_abort()`** callback methods are invoked.

## vmm_log_format::continue_msg()

Formats the continuation of a message.

### SystemVerilog

```
virtual function string continue_msg(
    string name,
    string instance,
    string msg_typ,
    string severity,
    ref string lines[$]);

With +define VMM_LOG_FORMAT_FILE_LINE

virtual function string continue_msg(
     string name,
     string inst,
     string msg_typ,
     string severity,
     string fname,
     int    line,
     ref string lines[$]);
```

### OpenVera

```
virtual function string continue_msg(string name,
     string instance,
     string msg_typ,
     string severity,
     string lines[$]);
```

### Description

This method is called by all message service interfaces to format the continuation of a message, and subsequent calls to the **`vmm_log::end_msg()`** method or empty **`vmm_log::text("")`**

method call. The first call to the **`vmm_log::end_msg()`** method or empty **`vmm_log::text("")`** method uses the **`vmm_log_format::format_msg()`** method.

A message on subsequent occurrences of a call to the `"vmm_log::end_msg()"` method or empty `"vmm_log::text()"` method call after a call to `"vmm_log::start_msg()"`. The first call to these methods call the `"vmm_log_format::format_msg()"` method.

For backward compatibility when using SystemVerilog, the `` `VMM_LOG_FORMAT_FILE_LINE `` symbol must be defined to enable the inclusion of the filename and line number to the message formatter.

## Example

*Example A-113*

```
...
string line[$];
string str;
super.build();
str = "Continue Msg string";

for(int idx = 0; idx < 5 ; idx++)
  line.push_back(str);
`vmm_note(log,$psprintf("%0s",this.format.continue_msg
              ("msg","log","","DEBUG_SEV",line)));
...
```

## vmm_log_format::format_msg()

Formats a message.

### SystemVerilog

```
virtual function string format_msg(
    string name,
     string instance,
     string msg_typ,
     string severity,
     ref string lines[$]);

With +define VMM_LOG_FORMAT_FILE_LINE

virtual function string format_msg(string name,
     string inst,
     string msg_typ,
     string severity,
     string fname,
     int    line,
     ref string lines[$]);
```

### OpenVera

```
virtual function string format_msg(string name,
     string instance,
     string msg_typ,
     string severity,
     string lines[$]);
```

### Description

Returns a fully formatted image of the message, as specified by the arguments. The **`lines`** parameter contains one line of message text for each non-empty call to the **`vmm_log::text()`** method. A line may contain newline characters.

This method is called by all message service interfaces to format a message on the first occurrence of a call to the **vmm_log::end_msg()** method or empty **vmm_log::text()** method call after a call to **vmm_log::start_msg()**. Subsequent calls to these methods call the **vmm_log_format::continue_msg()** method.

For backward compatibility when using SystemVerilog, the **`VMM_LOG_FORMAT_FILE_LINE** symbol must be defined to enable the inclusion of the filename and line number to the message formatter.

## Example

*Example A-114*

```
class env_log_fmt extends vmm_log_format;
   function string format_msg(string name = "", string
instance = "",
                           string msg_type,  string severity,
                                ref string lines[$]);
   for(int i=0;i<lines.size;i++)
      $sformat(format_msg,
       "%0t, (%s) (%s) [%0s:%0s] \n \t \t %s ",
       $time, name, instance, msg_type, severity, lines[i]);
   endfunction
endclass

class my_env extends vmm_env;
   ...
   env_log_fmt  env_fmt = new();
   function new();
      this.log.set_format(env_fmt);
      `vmm_note(log,"Inside New");
   endfunction
endclass
```

## vmm_log_format::pass_or_fail()

Formats the final pass or fail message at the end of simulation.

### SystemVerilog

```
virtual function string pass_or_fail(bit pass,
    string name,
    string inst,
    int fatals,
    int errors,
    int warnings,
    int dem_errs,
    int dem_warns);
```

### OpenVera

Not supported.

### Description

This method is called by the **`vmm_log::report()`** method to format the final pass or fail message, at the end of simulation.

The **`pass`** argument, if true, indicates that the simulation was successful.

The **`name`** and **`instance`** arguments are the specified name and instance names specified to the **`vmm_log::report()`** method.

The **`fatals`** argument is the total number of **`vmm_log::FATAL_SEV`** messages that were generated.

The **`errors`** argument is the total number of **`vmm_log::ERROR_SEV`** messages that were generated.

The **warnings** argument is the total number of **vmm_log::WARNING_SEV** messages that were generated.

The **dem_errs** argument is the total number of **vmm_log::ERROR_SEV** messages that were demoted.

The **dem_warns** argument is the total number of **vmm_log::WARNING_SEV** messages that were demoted.

# vmm_ms_scenario

This is a base class for all user-defined multi-stream scenario descriptors. This class extends from vmm_scenario.

## Summary


- vmm_ms_scenario::execute() ..... page A-342
- vmm_ms_scenario::get_channel() ..... page A-344
- vmm_ms_scenario::get_context_gen() ..... page A-346
- vmm_ms_scenario::get_ms_scenario() ..... page A-348
- vmm_ms_scenario::new() ..... page A-350

## vmm_ms_scenario::execute()

Executes a multi-stream scenario.

### SystemVerilog

```
virtual task execute(ref int n)
```

### OpenVera

Not supported.

### Description

Execute the scenario. Increments the argument "*n*" by the total number of transactions that were executed in this scenario.

This method must be overloaded to procedurally define a multi-stream scenario.

### Example

*Example A-115*

```
class my_scenario extends vmm_ms_scenario;
   my_atm_cell_scenario atm_scenario;
   my_cpu_scenario cpu_scenario;
   ...
   function new;
      super.new(null);
      atm_scenario = new;
      cpu_scenario = new;
   endfunction: new

   task execute(ref int n);
      fork
        begin
```

```
            atm_cell_channel out_chan;
            int unsigned nn = 0;
            $cast(out_chan, this.get_channel(
                 "ATM_SCENARIO_CHANNEL"));
            atm_scenario.randomize with {length == 1;};
            atm_scenario.apply(out_chan, nn);
            n += nn;
          end
          begin
            cpu_channel out_chan;
            int unsigned nn = 0;
            $cast(out_chan,this.get_channel(
                "CPU_SCENARIO_CHANNEL"));
            cpu_scenario.randomize with {length == 1;};
            cpu_scenario.apply(out_chan, nn);
            n += nn;
          end
        join
    endtask: execute
    ...
endclass: my_scenario
```

## vmm_ms_scenario::get_channel()

Returns a registered output channel.

### SystemVerilog

```
function vmm_channel get_channel(string name)
```

### OpenVera

Not supported.

### Description

Returns the output channel, which is registered under the specified logical name in the multi-stream generator where the multi-stream scenario generator is registered. Returns `NULL`, if no such channel exists.

### Example

*Example A-116*

```
`vmm_channel(atm_cell)
`vmm_scenario_gen(atm_cell, "atm trans")

program test_ms_scenario;
   vmm_ms_scenario_gen atm_ms_gen =
      new("Atm Scenario Gen", 12);
   atm_cell_channel my_chan=new("MY_CHANNEL", "EXAMPLE");
   atm_cell_channel buffer_channel = new("MY_BUFFER",
"EXAMPLE");
   ...
   initial begin
      ...
     buffer_channel = atm_ms_gen.get_channel("MY_CHANNEL");
      if(buffer_channel != null)
```

```
         vmm_log(log,"Returned channel \n");
         ...
      else
         vmm_log(log,"Returned null value\n");
      ...
   end

endprogram
```

## vmm_ms_scenario::get_context_gen()

Returns the multi-stream scenario generator that is executing this scenario.

### SystemVerilog

```
function vmm_ms_scenario_gen get_context_gen()
```

### OpenVera

Not supported.

### Description

Returns a reference to the multi-stream scenario generator that is providing the context for the execution of this multi-stream scenario descriptor. Returns `NULL`, if this multi-stream scenario descriptor is not registered with a multi-stream scenario generator.

### Example

*Example A-117*

```
`vmm_scenario_gen(atm_cell, "atm trans")

program test_ms_scenario;
   vmm_ms_scenario_gen atm_ms_gen =
       new("Atm Scenario Gen", 12);
   vmm_ms_scenario ms_scen = new;
   ...
   initial begin
      atm_ms_gen.register_ms_scenario(
          "FIRST SCEN",first_ms_scen);
      ...
      if(my_scen.get_context_gen())
```

```
        vmm_log(log,"This scenario has been registered\n");
          ...
      else
         vmm_log(log,"Scenario not yet registered \n");
      ...
   end
endprogram
```

## vmm_ms_scenario::get_ms_scenario()

Returns a registered multi-stream scenario descriptor.

### SystemVerilog

```
function vmm_ms_scenario get_ms_scenario(string scenario,
    string gen = "")
```

### OpenVera

Not supported.

### Description

Returns a copy of the multi-stream scenario that is registered under the specified scenario name, in the multi-stream generator that is registered under the specified generator name. Returns `NULL`, if no such scenario exists. Therefore, `vmm_ms_scenario::copy()` should be overloaded for multistream scenarios to return the copy of the scenario.

If no generator name is specified, searches the scenario registry of the generator that is executing this scenario.

The scenario can then be executed within the context of the generator where it is registered by calling its vmm_ms_scenario::execute() method.

### Example

*Example A-118*

```
`vmm_scenario_gen(atm_cell, "atm trans")
program test_ms_scenario;
```

```
   vmm_ms_scenario_gen atm_ms_gen =
      new("Atm Scenario Gen", 12);
   vmm_ms_scenario first_ms_scen = new;
   vmm_ms_scenario buffer_ms_scen = new;
   ...
   initial begin
      atm_ms_gen.register_ms_scenario("FIRST
SCEN",first_ms_scen);
      ...
      buffer_ms_scen = atm_ms_gen.get_ms_scenario("FIRST
SCEN");
      if(buffer_ms_scen != null)
         vmm_log(log,"Returned scenario \n");
         ...
      else
         vmm_log(log,"Returned null, scenario doesn't
exists\n");
      ...
   end

endprogram
```

## vmm_ms_scenario::new()

Instantiates a multi-stream scenario descriptor.

### SystemVerilog

```
function new(vmm_scenario parent = null)
```

### OpenVera

Not supported.

### Description

Creates a new instance of a multi-stream scenario descriptor.

If a parent scenario descriptor is specified, then this instance of a multi-stream scenario descriptor is assumed to be instantiated inside the specified scenario descriptor, creating a hierarchical multi-stream scenario descriptor.

If no parent scenario descriptor is specified, then it is assumed to be a top-level scenario descriptor.

### Example

*Example A-119*

```
class my_scenario extends vmm_ms_scenario;
   function new;
      super.new(null);
   endfunction: new
endclass
program test;
   my_scenario sc0 = new;
endprogram
```

## vmm_ms_scenario_gen

This class is a pre-defined multi-stream scenario generator.

VMM provides this class to model general purpose scenarios. It is possible to generate heterogeneous scenarios, and have them controlled by a unique transactor.

The multi-stream scenario generation mechanism provides an efficient way of generating and synchronizing stimulus to various BFMs. This helps you to reuse block level scenarios in subsystem and system levels, and controlling or synchronizing the execution of those scenarios of same or different streams. Single stream scenarios can also be reused in multi-stream scenarios. **`vmm_ms_scenario`** and **`vmm_ms_scenario_gen`** are the base classes provided by VMM for this functionality. This section describes the various usages of multi-stream scenario generation with these base classes.

Generated scenarios can be transferred to any number of channels of various types, anytime during simulation, making this solution very scalable, dynamic and completely controllable. Moreoverer, it is possible to model sub-scenarios that can be attached and controlled by an overall scenario, in a hierarchical way. You can determine the number of scenarios or the number of transactions to be generated, either on a MSS basis or on a given scenario generator, making this use model scalable from block to system level.

It is also possible to add or remove scenarios as simulation advances, facilitating detection of corner cases or address other constraints on the fly. If multiple scenario generators should access a common channel, then it is possible to give the channel access to

only one generator on a given time slot. In this case, other generators do wait until the channel is released, thereby making it a blocking transaction.

The following methods are available.

## Summary


- vmm_ms_scenario_gen::channel_exists() ..... page A-353
- vmm_ms_scenario_gen::DONE ..... page A-355
- vmm_ms_scenario_gen::GENERATED ..... page A-356
- vmm_ms_scenario_gen::get_all_channel_names() ..... page A-357
- vmm_ms_scenario_gen::get_all_ms_scenario_names() . page A-358
- vmm_ms_scenario_gen::get_all_ms_scenario_gen_names() page A-359
- vmm_ms_scenario_gen::get_channel() ..... page A-360
- vmm_ms_scenario_gen::get_channel_name() ..... page A-362
- vmm_ms_scenario_gen::get_ms_scenario_index() ..... page A-363
- vmm_ms_scenario_gen::get_ms_scenario() ..... page A-365
- vmm_ms_scenario_gen::get_ms_scenario_gen() ..... page A-366
- vmm_ms_scenario_gen::get_ms_scenario_gen_name() .. page A-368
- vmm_ms_scenario_gen::get_ms_scenario_name() ..... page A-369
- vmm_ms_scenario_gen::get_n_insts() ..... page A-371
- vmm_ms_scenario_gen::get_n_scenarios() ..... page A-372
- vmm_ms_scenario_gen::get_names_by_channel() ..... page A-373
- vmm_ms_scenario_gen::get_names_by_ms_scenario() .. page A-375
- vmm_ms_scenario_gen::get_names_by_ms_scenario_gen() page A-377
- vmm_ms_scenario_gen::inst_count ..... page A-379
- vmm_ms_scenario_gen::ms_scenario_exists() ..... page A-381
- vmm_ms_scenario_gen::ms_scenario_gen_exists() .... page A-383
- vmm_ms_scenario_gen::register_channel() ..... page A-385
- vmm_ms_scenario_gen::register_ms_scenario() ..... page A-387
- vmm_ms_scenario_gen::register_ms_scenario_gen() .. page A-389
- vmm_ms_scenario_gen::replace_channel() ..... page A-391
- vmm_ms_scenario_gen::replace_ms_scenario() ..... page A-393
- vmm_ms_scenario_gen::replace_ms_scenario_gen() ... page A-395
- vmm_ms_scenario_gen::scenario_count ..... page A-396
- vmm_ms_scenario_gen::scenario_set[$] ..... page A-398
- vmm_ms_scenario_gen::select_scenario ..... page A-400
- vmm_ms_scenario_gen::stop_after_n_insts ..... page A-402
- vmm_ms_scenario_gen::stop_after_n_scenarios ..... page A-404
- vmm_ms_scenario_gen::unregister_channel() ..... page A-406
- vmm_ms_scenario_gen::unregister_channel_by_name() page A-408
- vmm_ms_scenario_gen::unregister_ms_scenario() .... page A-410
- vmm_ms_scenario_gen::unregister_ms_scenario_by_name() page A-412
- vmm_ms_scenario_gen::unregister_ms_scenario_gen() page A-414
- vmm_ms_scenario_gen::unregister_ms_scenario_gen_by_name() page A-416

## vmm_ms_scenario_gen::channel_exists()

Checks if a channel is registered under a specified name.

### SystemVerilog

```
virtual function bit channel_exists(string name)
```

### OpenVera

Not supported.

### Description

Returns `TRUE`, if there is an output channel registered under the specified name. Otherwise, it returns `FALSE`.

Use the `vmm_ms_scenario_gen::get_channel()` method to retrieve a channel under a specified name.

### Example

*Example A-120*

```
`vmm_channel(atm_cell)
`vmm_scenario_gen(atm_cell, "atm_trans")

program test_scen;
   vmm_ms_scenario_gen my_ms_gen =
      new("MS Scenario Gen", 11);
   atm_cell_channel ms_chan_1 =
      new("MS-CHANNEL-1", "MY_CHANNEL");
   ...
   initial begin
      vmm_log(log,"Registering channel \n");
     my_ms_gen.register_channel("MS-CHANNEL-1",ms_chan_1);
      ...
```

```
      if(my_ms_gen.channel_exists("MS_CHANNEL-1"))
         vmm_log(log,"Channel exists\n");
      else
         vmm_log(log,"Channel not yet registered\n");
      ...
   end
endprogram
```

## vmm_ms_scenario_gen::DONE

Notifiies the completed generation.

### SystemVerilog

```
typedef enum int {DONE} symbols_e
```

### OpenVera

Not supported.

### Description

Notification in `vmm_xactor::notify` that is indicated when the generation process is completed, as specified by the `vmm_ms_scenario_gen::stop_after_n_scenarios` and `vmm_ms_scenario_gen::stop_after_n_insts` class properties.

### Example

*Example A-121*

```
program test_scen;
   ...
   vmm_ms_scenario_gen my_ms_gen = new(
      "MY MS SCENARIO",10);
      initial begin
      ...
      `vmm_note(log,"Waiting for notification : DONE \n");
      my_ms_gen.notify.wait_for(
         vmm_ms_scenario_gen::DONE);
      ...
   end
end
```

## vmm_ms_scenario_gen::GENERATED

Notifies the newly generated scenario.

### SystemVerilog

```
typedef enum int {GENERATED} symbols_e
```

### OpenVera

Not supported.

### Description

Notification in `vmm_xactor::notify` that is indicated, every time a new multi-stream scenario is generated and about to be executed.

### Example

*Example A-122*

```
program test_scen;
   ...
   vmm_ms_scenario_gen my_ms_gen= new("MY MS SCENARIO",10);
   ...
   initial begin
      ...
      `vmm_note(
         log,"Waiting for notification : GENERATED \n");
      my_ms_gen.notify.wait_for(
         vmm_ms_scenario_gen::GENERATED);
      ...
   end
end
```

## vmm_ms_scenario_gen::get_all_channel_names()

Returns all names in the channel registry.

### SystemVerilog

```
virtual function void get_all_channel_names(
     ref string name[$])
```

### OpenVera

Not supported.

### Description

Appends the names under which an output channel is registered. Returns the number of names that were added to the array.

### Example

*Example A-123*

```
`vmm_channel(atm_cell)
`vmm_scenario_gen(atm_cell, "atm_trans")

program test_scen;
   vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen",
11);
   atm_cell_channel ms_chan_1=new("MS-CHANNEL-1",
"MY_CHANNEL");
   string channel_name_array[$];
   ...
   initial begin
      `vmm_note(log,"Registering channel \n");
     my_ms_gen.register_channel("MS-CHANNEL-1",ms_chan_1);
      my_ms_gen.get_all_channel_names(channel_name_array);
   end
endprogram
```

## vmm_ms_scenario_gen::get_all_ms_scenario_names()

Returns all names in the scenario registry.

### SystemVerilog

```
virtual function void get_all_ms_scenario_names(
          ref string name[$])
```

### OpenVera

Not supported.

### Description

Appends the names under which a multi-stream scenario descriptor is registered. Returns the number of names that were added to the array.

### Example

*Example A-124*

```
class my_ms_scen extends vmm_ms_scenario;
endclass
program test_scenario;
   string scen_name_arr[$];
  vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen", 9);
   my_ms_scen ms_scen = new;
   initial begin
      `vmm_note(log,"Registering MS scenario \n");
      my_ms_gen.register_ms_scenario("MS-SCEN-1",ms_scen);
      my_ms_gen.register_ms_scenario("MS-SCEN-2",ms_scen);
    my_ms_gen.get_all_ms_scenario_names(scen_name_arr);
   end
endprogram
```

## vmm_ms_scenario_gen::get_all_ms_scenario_gen_names()

Returns all names in the generator registry.

### SystemVerilog

```
virtual function void get_all_ms_scenario_gen_names(
    ref string name[$])
```

### OpenVera

Not supported.

### Description

Appends the names under which a sub-generator is registered. Returns the number of names that were added to the array.

### Example

*Example A-125*

```
program test_scenario;
   string ms_gen_names_arr[$];
   vmm_ms_scenario_gen parent_ms_gen =
      new("Parent-MS-Scen-Gen", 11);
   vmm_ms_scenario_gen child_ms_gen  =
      new("Child-MS-Scen-Gen", 6);
   ...
   initial begin
      `vmm_note(log,"Registering sub MS generator \n");
      parent_ms_gen.register_ms_scenario_gen(
         "Child-MS-Scen-Gen",child_ms_gen);
      parent_ms_gen.get_all_ms_scenario_gen_names(
         ms_gen_names_arr);
   end
endprogram
```

## vmm_ms_scenario_gen::get_channel()

Returns the channel that is registered under a specified name.

### SystemVerilog

```
virtual function vmm_channel get_channel(
     string name)
```

### OpenVera

Not supported.

### Description

Returns the output channel registered under the specified name. Generates a warning message and returns `NULL`, if there are no channels registered under that name.

### Example

*Example A-126*

```
`vmm_channel(atm_cell)
`vmm_scenario_gen(atm_cell, "atm_trans")

program test_scen;
     vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen",
11);
     atm_cell_channel ms_chan_1=new("MS-CHANNEL-1",
"MY_CHANNEL");
     atm_cell_channel buffer_chan = new("BUFFER","MY_BC");
     ...
     initial begin
          vmm_log(log,"Registering channel \n");
          my_ms_gen.register_channel("MS-CHANNEL-
1",ms_chan_1);
```

```
          ...
          buffer_chan = my_ms_gen.get_channel("MS-CHANNEL-
1");
          ...
     end
endprogram
```

## vmm_ms_scenario_gen::get_channel_name()

Returns a name under which a channel is registered.

### SystemVerilog

```
virtual function string get_channel_name(vmm_channel chan)
```

### OpenVera

Not supported.

### Description

Return a name under which the specified channel is registered. Returns `""`, if the channel is not registered.

### Example

*Example A-127*

```
`vmm_channel(atm_cell)
`vmm_scenario_gen(atm_cell, "atm_trans")

program test_scen;
     vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen",
11);
     atm_cell_channel ms_chan_1=new("MS-CHANNEL-1",
"MY_CHANNEL");
     string buffer_chan_name;
     initial begin
          vmm_log(log,"Registering channel \n");
     my_ms_gen.register_channel("MS-CHANNEL-1",ms_chan_1);
          buffer_chan_name =
my_ms_gen.get_channel_name(ms_chan_1);
     end
endprogram
```

## vmm_ms_scenario_gen::get_ms_scenario_index()

Returns the index of the specified scenario.

### SystemVerilog

```
virtual function int get_ms_scenario_index(
    vmm_ms_scenario scenario)
```

### OpenVera

Not supported.

### Description

Returns the index of the specified scenario descriptor in the vmm_ms_scenario_gen::scenario_set[$] array. A warning message is generated and returns -1, if the scenario descriptor is not found in the scenario set.

### Example

*Example A-128*

```
class my_ms_scen extends vmm_ms_scenario;
   ...
endclass

program test_scenario;
  vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen", 9);
   my_ms_scen ms_scen = new;
   int buffer_index;

   initial begin
      ...
      vmm_log(log,"Registering MS scenario \n");
      my_ms_gen.register_ms_scenario("MS-SCEN-1",ms_scen);
```

```
      ...
      buffer_index =
         my_ms_gen.get_ms_scenario_index(ms_scen);
      vmm_note(log,`vmm_sformatf(
         "Index for ms_scen is : %d\n",buffer_index));
      ...
   end
endprogram
```

## vmm_ms_scenario_gen::get_ms_scenario()

Returns the scenario that is registered under a specified name.

### SystemVerilog

```
virtual function vmm_ms_scenario get_ms_scenario(
     string name)
```

### OpenVera

Not supported.

### Description

Returns a copy of the multi-stream scenario descriptor that is registered under the specified name. Generates a warning message and returns `NULL`, if there are no scenarios registered under that name.

### Example

*Example A-129*

```
class my_ms_scen extends vmm_ms_scenario;
endclass

program test_scenario;
  vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen", 9);
   my_ms_scen ms_scen = new;
   my_ms_scen buffer_scen = new;
   initial begin
      vmm_log(log,"Registering MS scenario \n");
      my_ms_gen.register_ms_scenario("MS-SCEN-1",ms_scen);
     buffer_scen = my_ms_gen.get_ms_scenario("MY-SCEN_1");
   end
endprogram
```

## vmm_ms_scenario_gen::get_ms_scenario_gen()

Returns the sub-generator that is registered under a specified name.

### SystemVerilog

```
virtual function vmm_ms_scenario_gen get_ms_scenario_gen(
     string name)
```

### OpenVera

Not supported.

### Description

Returns the sub-generator that is registered under the specified name. Generates a warning message and returns `NULL`, if there are no generators registered under that name.

### Example

*Example A-130*

```
program test_scenario;
   vmm_ms_scenario_gen parent_ms_gen =
      new("Parent-MS-Scen-Gen", 11);
   vmm_ms_scenario_gen child_ms_gen  =
      new("Child-MS-Scen-Gen", 6);
   vmm_ms_scenario_gen buffer_ms_gen =
      new("Buffer-MS-Scen-Gen", 6);
   ...
   initial begin
      vmm_log(log,"Registering sub MS generator \n");
      parent_ms_gen.register_ms_scenario_gen(
         "Child-MS-Scen-Gen", child_ms_gen);
      ...
      buffer_ms_gen = parent_ms_gen.get_ms_scenario_gen(
```

```
      "Child-MS-Scen-Gen");
   ...
   end
endprogram
```

## vmm_ms_scenario_gen::get_ms_scenario_gen_name()

Returns a name under which a generator is registered.

### SystemVerilog

```
virtual function string get_ms_scenario_gen_name(
     vmm_ms_scenario_gen scenario_gen)
```

### OpenVera

Not supported.

### Description

Returns a name under which the specified sub-generator is registered. Returns "", if the generator is not registered.

### Example

*Example A-131*

```
program test_scenario;
   string buffer_ms_gen_name;
   vmm_ms_scenario_gen parent_ms_gen =
      new("Parent-MS-Scen-Gen", 11);
   vmm_ms_scenario_gen child_ms_gen  =
      new("Child-MS-Scen-Gen", 6);
   initial begin
      vmm_log(log,"Registering sub MS generator \n");
      parent_ms_gen.register_ms_scenario_gen(
         "Child-MS-Scen-Gen",child_ms_gen);
      buffer_ms_gen_name =
         parent_ms_gen.get_ms_scenario_gen_name(
         child_ms_gen);
   end
endprogram
```

## vmm_ms_scenario_gen::get_ms_scenario_name()

Returns a name under which a scenario is registered.

### SystemVerilog

```
virtual function string get_ms_scenario_name(
    vmm_ms_scenario scenario)
```

### OpenVera

Not supported.

### Description

Returns a name under which the specified multi-stream scenario descriptor is registered. Returns "", if the scenario is not registered.

### Example

*Example A-132*

```
class my_ms_scen extends vmm_ms_scenario;
   ...
endclass

program test_scenario;
  vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen", 9);
   my_ms_scen ms_scen = new;
   string buffer_name;

   initial begin
      ...
      vmm_log(log,"Registering MS scenario \n");
      my_ms_gen.register_ms_scenario("MS-SCEN-1",ms_scen);
      ...
     buffer_name = my_ms_gen.get_ms_scenario_name(ms_scen);
```

```
      vmm_note(log,
         `vmm_sformatf(
            "Registered name for ms_scen is: %s\n",
            buffer_name));
      ...
   end
endprogram
```

## vmm_ms_scenario_gen::get_n_insts()

Returns the number of transaction descriptors generated so far.

### SystemVerilog

```
function int unsigned get_n_insts()
```

### OpenVera

Not supported.

### Description

Returns the current value of the vmm_ms_scenario_gen::inst_count property.

### Example

*Example A-133*

```
class my_ms_scen extends vmm_ms_scenario_gen;
   ...
   function void print_ms_gen_fields();
      ...
      `vmm_note(log,$psprintf(
         "Present instance count is %d\n",
         this.get_n_insts()));
   endfunction
endclass

program test_scen;
   my_ms_scen my_gen= new("MY MS SCENARIO",10);
   initial begin
   my_gen.print_ms_gen_fields();
   end
end
```

## vmm_ms_scenario_gen::get_n_scenarios()

Returns the number of multi-stream scenarios generated so far.

### SystemVerilog

```
function int unsigned get_n_scenarios()
```

### OpenVera

Not supported.

### Description

Returns the current value of the `vmm_ms_scenario_gen::scenario_count` property.

### Example

*Example A-134*

```
class my_ms_scen extends vmm_ms_scenario_gen;
   ...
   function void print_ms_gen_fields();
      ...
      `vmm_note(log,$psprintf(
          "Present scenario count is %d\n",
          this.get_n_scenarios()));
   endfunction
endclass

program test_scen;
   my_ms_scen my_gen= new("MY MS SCENARIO",10);
   initial begin
   my_gen.print_ms_gen_fields();
   end
end
```

## vmm_ms_scenario_gen::get_names_by_channel()

Returns the names under which a channel is registered.

### SystemVerilog

```
virtual function void get_names_by_channel(
     vmm_channel chan,
     ref string   name[$])
```

### OpenVera

Not supported.

### Description

Appends the names under which the specified output channel is registered. Returns the number of names that were added to the array.

### Example

*Example A-135*

```
`vmm_channel(atm_cell)
`vmm_scenario_gen(atm_cell, "atm_trans")

program test_scen;
   vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen",
11);
   atm_cell_channel ms_chan_1=new("MS-CHANNEL-1",
"MY_CHANNEL");
   string channel_name_array[$];
   ...
   initial begin
      `vmm_note(log,"Registering channel \n");
     my_ms_gen.register_channel("MS-CHANNEL-1",ms_chan_1);
```

```
      ...

my_ms_gen.get_names_by_channel(ms_chan_1,channel_name_arra
y);
      ...
   end
endprogram
```

## vmm_ms_scenario_gen::get_names_by_ms_scenario()

Returns the names under which a scenario is registered.

### SystemVerilog

```
virtual function void get_names_by_ms_scenario(
     vmm_ms_scenario scenario,
     ref string       name[$])
```

### OpenVera

Not supported.

### Description

Appends the names under which the specified multi-stream scenario descriptor is registered. Returns the number of names that were added to the array.

### Example

*Example A-136*

```
class my_ms_scen extends vmm_ms_scenario;
   ...
endclass

program test_scenario;
   string scen_name_arr[$];
  vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen", 9);
   my_ms_scen ms_scen = new;
   ...
   initial begin
      ...
      `vmm_note(log,"Registering MS scenario \n");
      my_ms_gen.register_ms_scenario("MS-SCEN-1",ms_scen);
```

```
      my_ms_gen.register_ms_scenario("MS-SCEN-2",ms_scen);
      ...
      my_ms_gen.get_names_by_ms_scenario(
         ms_scen,scen_name_arr);
      ...
   end
endprogram
```

## vmm_ms_scenario_gen::get_names_by_ms_scenario_gen()

Returns the names under which a generator is registered.

### SystemVerilog

```
virtual function void
    get_names_by_ms_scenario_gen(vmm_ms_scenario_gen
scenario_gen, ref string name[$])
```

### OpenVera

Not supported.

### Description

Appends the names under which the specified sub-generator is registered. Returns the number of names that were added to the array.

### Example

*Example A-137*

```
program test_scenario;
   string ms_gen_names_arr[$];
   vmm_ms_scenario_gen parent_ms_gen =
       new("Parent-MS-Scen-Gen", 11);
   vmm_ms_scenario_gen child_ms_gen  =
       new("Child-MS-Scen-Gen", 6);
   ...
   initial begin
      `vmm_note(log,"Registering sub MS generator \n");
      parent_ms_gen.register_ms_scenario_gen(
         "Child-MS-Scen-Gen", child_ms_gen);
      ...
```

```
parent_ms_gen.get_names_by_ms_scenario_gen(child_ms_gen,
     ms_gen_names_arr);
       ...
   end

endprogram
```

## vmm_ms_scenario_gen::inst_count

Returns the number of transaction descriptor generated so far.

### SystemVerilog

```
protected int inst_count;
```

### OpenVera

Not supported.

### Description

Returns the current count of the number of individual transaction descriptor instances generated by the multi-stream scenario generator. When it reaches or surpasses the value in `vmm_ms_scenario_gen::stop_after_n_insts`, the generator stops.

The number of transaction descriptor instances generated by the execution of a multi-stream scenario is the number of transactions reported by the `vmm_ms_scenario::execute()` method, when it returns.

### Example

*Example A-138*

```
class my_ms_scen extends vmm_ms_scenario_gen;
   ...
   function void print_ms_gen_fields();
      ...
      `vmm_note(log,$psprintf(
       "Present instance count is %d\n", this.inst_count));
```

```
   endfunction
   ...
endclass

program test_scen;
   ...
   my_ms_scen my_gen= new("MY MS SCENARIO",10);
   ...
   initial begin
   ...
   my_gen.print_ms_gen_fields();
   ...
   end
end
```

## vmm_ms_scenario_gen::ms_scenario_exists()

Checks if a scenario is registered under a specified name.

### SystemVerilog

```
virtual function bit ms_scenario_exists(string name)
```

### OpenVera

Not supported.

### Description

Returns `TRUE`, if there is a multi-stream scenario registered under the specified name. Otherwise, it returns `FALSE`.

Use the `vmm_ms_scenario_gen::get_ms_scenario()` method to retrieve a scenario under a specified name.

### Example

*Example A-139*

```
class my_ms_scen extends vmm_ms_scenario;
   ...
endclass

program test_scenario;
  vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen", 9);
   my_ms_scen ms_scen = new;
   ...
   initial begin
      ...
      vmm_log(log,"Registering MS scenario \n");
      my_ms_gen.register_ms_scenario("MS SCEN-1",ms_scen);
      ...
```

```
      if(my_ms_gen.ms_scenario_exists("MS-SCEN-1"))
        `vmm_note(log, "Scenario MS-SCEN-1 is registered");
      else
         `vmm_note(log,
            "Scenario MS-SCEN-1 is not yet registered");
      ...
   end
endprogram
```

## vmm_ms_scenario_gen::ms_scenario_gen_exists()

Checks if a generator is registered under a specified name.

### SystemVerilog

```
virtual function bit ms_scenario_gen_exists(string name)
```

### OpenVera

Not supported.

### Description

Returns `TRUE`, if there is a sub-generator registered under the specified name. Otherwise, it returns `FALSE`.

Use the `vmm_ms_scenario_gen::get_ms_scenario_gen()` to retrieve a sub-generator under a specified name.

### Example

*Example A-140*

```
program test_scen;
   vmm_ms_scenario_gen parent_ms_gen =
      new("Parent-MS-Scen-Gen", 11);
   vmm_ms_scenario_gen child_ms_gen =
      new(" Child-MS-Scen-Gen", 6);
   ...
   initial begin
      vmm_log(log,"Registering sub MS generator \n");
      parent_ms_gen.register_ms_scenario_gen(
         "Child-MS-Scen-Gen",child_ms_gen);
      ...
      if(parent_ms_gen.ms_scenario_gen_exists(
            "Child-MS-Scen-Gen"))
```

```
            `vmm_note(log, "Generator exists in registry");
         else
            `vmm_note(log,
                "Generator doesn't exist in registry");
         ...
      end
   endprogram
```

## vmm_ms_scenario_gen::register_channel()

Registers an output channel.

### SystemVerilog

```
virtual function void register_channel(string name,
     vmm_channel chan)
```

### OpenVera

Not supported.

### Description

Registers the specified output channel under the specified logical name. The same channel may be registered multiple times under different names, thus creating an alias to the same channel.

Once registered, the output channel is available under the specified logical name to multi-stream scenarios through the `vmm_ms_scenario::get_channel()` method.

It is an error to register a channel under a name that already exists. Use the `vmm_ms_scenario_gen::replace_channel()` to replace a registered scenario.

### Example

*Example A-141*

```
`vmm_channel(atm_cell)
`vmm_scenario_gen(atm_cell, "atm_trans")

program test_scen;
```

```
   vmm_ms_scenario_gen my_ms_gen =
      new("MS Scenario Gen", 11);
   atm_cell_channel ms_chan_1 =
      new("MS-CHANNEL-1", "MY_CHANNEL");
   ...
   initial begin
      ...
      vmm_log(log,"Registering channel \n");
     my_ms_gen.register_channel("MS-channel-1",ms_chan_1);
      ...
   end
endprogram
```

## vmm_ms_scenario_gen::register_ms_scenario()

Registers a multi-stream scenario descriptor

### SystemVerilog

```
virtual function void register_ms_scenario(string name,
    vmm_ms_scenario scenario)
```

### OpenVera

Not supported.

### Description

Registers the specified multi-stream scenario under the specified name. The same scenario may be registered multiple times under different names, thus creating an alias to the same scenario.

Registering a scenario implicitly appends it to the scenario set, if it is not already in the `vmm_ms_scenario_gen::scenario_set[$]` array.

It is an error to register a scenario under a name that already exists. Use the `vmm_ms_scenario_gen::replace_ms_scenario()` to replace a registered scenario.

### Example

*Example A-142*

```
class my_ms_scen extends vmm_ms_scenario;
    ...
endclass
```

```
program test_scenario;
  vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen", 9);
   my_ms_scen ms_scen = new;
     ...
   initial begin
      ...
      vmm_log(log,"Registering MS scenario \n");
      my_ms_gen.register_ms_scenario("MS SCEN-1",ms_scen);
      ...
    end
endprogram
```

## vmm_ms_scenario_gen::register_ms_scenario_gen()

Registers a sub-generator

### SystemVerilog

```
virtual function void register_ms_scenario_gen(string name,
     vmm_ms_scenario_gen scenario_gen)
```

### OpenVera

Not supported.

### Description

Registers the specified sub-generator under the specified logical name. The same generator may be registered multiple times under different names, therefore creating an alias to the same generator.

Once registered, the multi-stream generator becomes available under the specified logical name to multi-stream scenarios via the `vmm_ms_scenario::get_ms_scenario()` method to create hierarchical multi-stream scenarios.

It is an error to register a generator under a name that already exists. Use the `vmm_ms_scenario_gen::replace_ms_scenario_gen()` method to replace a registered generator.

### Example

*Example A-143*

```
program test_scen;
     vmm_ms_scenario_gen parent_ms_gen = new("Parent-MS-
```

```
Scen-Gen", 11);
     vmm_ms_scenario_gen child_ms_gen = new(" Child-MS-Scen-
Gen", 6);
     ...
     initial begin
          vmm_log(log,"Registering sub MS generator \n");
          parent_ms_gen.register_ms_scenario_gen("Child-MS-
Scen-


Gen",child_ms_gen);
          ...
     end
endprogram
```

## vmm_ms_scenario_gen::replace_channel()

Replaces an output channel.

### SystemVerilog

```
virtual function void replace_channel(string name,
     vmm_channel chan)
```

### OpenVera

Not supported.

### Description

Registers the specified output channel under the specified name, replacing the channel that is previously registered under that name (if any). The same channel may be registered multiple times under different names, thus creating an alias to the same output channel.

### Example

*Example A-144*

```
`vmm_channel(atm_cell)
`vmm_scenario_gen(atm_cell, "atm_trans")

program test_scen;
     vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen",
11);
     atm_cell_channel ms_chan_1=new("MS-CHANNEL-1",
"MY_CHANNEL");
     ...
     initial begin
          vmm_log(log,"Registering channel \n");
          my_ms_gen.register_channel("MS-CHANNEL-
1",ms_chan_1);
```

```
          my_ms_gen.register_channel("MS-CHANNEL-
2",ms_chan_1);
          ...
          vmm_log(log,"Replacing the channel \n");
          my_ms_gen.replace_channel("MS-CHANNEl-
1",ms_chan_1);
          ...
     end
endprogram
```

## vmm_ms_scenario_gen::replace_ms_scenario()

Replaces a scenario descriptor.

### SystemVerilog

```
virtual function void replace_ms_scenario(string name,
    vmm_ms_scenario scenario)
```

### OpenVera

Not supported.

### Description

Registers the specified multi-stream scenario under the specified name, replacing the scenario that is previously registered under that name (if any). The same scenario may be registered multiple times, under different names, thus creating an alias to the same scenario.

Registering a scenario implicitly appends it to the scenario set, if it is not already in the `vmm_ms_scenario_gen::scenario_set[$]` array. The replaced scenario is removed from the `vmm_ms_scenario_gen::scenario_set[$]` array, if it is not also registered under another name.

### Example

*Example A-145*

```
class my_ms_scen extends vmm_ms_scenario;
   ...
endclass

program test_scenario;
  vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen", 9);
```

```
   my_ms_scen ms_scen = new;
   ...
   initial begin
      ...
      my_ms_gen.register_ms_scenario("MS SCEN-1",ms_scen);
      my_ms_gen.register_ms_scenario("MS SCEN-2",ms_scen);
      ...
      vmm_log(log,"Replacing MS scenario \n");
      my_ms_gen.replace_ms_scenario("MS SCEN-1",ms_scen);
      ...
   end
endprogram
```

## vmm_ms_scenario_gen::replace_ms_scenario_gen()

Replaces a sub-generator.

### SystemVerilog

```
virtual function void replace_ms_scenario_gen(string name,
     vmm_ms_scenario_gen scenario_gen)
```

### OpenVera

Not supported.

### Description

Registers the specified sub-generator under the specified name, replacing the generator that is previously registered under that name (if any). The same generator may be registered multiple times under different names, thus creating an alias to the same sub-generator.

## vmm_ms_scenario_gen::scenario_count

Returns the number of multi-stream scenarios generated so far.

### SystemVerilog

```
protected int scenario_count;
```

### OpenVera

Not supported.

### Description

Returns the current count of the number of top-level multi-stream scenarios generated the multi-stream scenario generator. When it reaches or surpasses the value in `vmm_ms_scenario_gen::stop_after_n_scenarios`, the generator stops.

Only the multi-stream scenarios that are explicitly executed by this instance of the multi-stream scenario generator are counted. Sub-scenarios executed as part of a higher-level multi-stream scenario are not counted.

### Example

*Example A-146*

```
class my_ms_scen extends vmm_ms_scenario_gen;
   ...
   function void print_ms_gen_fields();
      ...
      `vmm_note(log,$psprintf(
         "Present scenario count is %d\n",
```

```
      this.scenario_count));
   endfunction
   ...
endclass

program test_scen;
   ...
   my_ms_scen my_gen= new("MY MS SCENARIO",10);
   ...
   initial begin
      fork
         begin
            @event;
            my_gen.print_ms_gen_fields();
         end
         ...
      join
      ...
   end
end
```

## vmm_ms_scenario_gen::scenario_set[$]

Multi-stream scenarios available for execution.

### SystemVerilog

```
vmm_ms_scenario scenatio_set[$]
```

### OpenVera

Not supported.

### Description

Multi-stream scenarios available for execution by this generator. The scenario executed next, is selected by randomizing the `vmm_ms_scenario_gen::select_scenario` class property.

Multi-stream scenario instances in this array should be managed through the `vmm_ms_scenario_gen::register_ms_scenario()`, `vmm_ms_scenario_gen::replace_ms_scenario()` and `vmm_ms_scenario_gen::unregister_ms_scenario()` methods.

### Example

*Example A-147*

```
class my_ms_scen extends vmm_ms_scenario;
   ...
endclass

program test_scenario;
   vmm_ms_scenario_gen parent_ms_gen =
```

```
      new("Parent-MS-Scen-Gen", 11);
   my_ms_scen ms_scen_1 = new;
   my_ms_scen ms_scen_2 = new;
   ...
   initial begin
      parent_ms_gen.register_ms_scenario(
         "MS-Scen-1",ms_scen_1);
      parent_ms_gen.register_ms_scenario(
         "MS-Scen-2",ms_scen_2);
      ...
      buffer_ms_gen =
         parent_ms_gen.unregister_ms_scenario(ms_scen_1);
      current_size = parent_ms_gen.scenario_set.size();
      `vmm_note(log, `vmm_sformatf(
       "Current size of scenario set is %d\n",current_size);
   end
endprogram
```

## vmm_ms_scenario_gen::select_scenario

Selects the scenario factory.

### SystemVerilog

```
vmm_ms_scenario_election select_scenario
```

### OpenVera

Not supported.

### Description

Randomly selects the next multi-stream scenario, to execute from the `vmm_ms_scenario_gen::scenario_set[$]` array. The selection is performed by calling the `randomize()` method on this class property, and then executing the multi-stream scenario found in the `vmm_ms_scenario_gen::scenario_set[$]` array at the index specified by the `vmm_ms_scenario_election::select` class property.

The default election instance may be replaced by a user-defined extension to modify the scenario election policy.

### Example

*Example A-148*

```
program test_scenario;
   vmm_ms_scenario_gen parent_ms_gen =
      new("Parent-MS-Scen-Gen", 11);
   my_ms_scen ms_scen_1 = new;
   ...
   initial begin
```

```
      parent_ms_gen.register_ms_scenario(
         "MS-Scen-1",ms_scen_1);
      ...
     parent_ms_gen.select_scenario.round_robin.constraint_
        mode(0);
      ...
   end
endprogram
```

## vmm_ms_scenario_gen::stop_after_n_insts

Returns the number of transaction descriptor to generate.

### SystemVerilog

```
int unsigned stop_after_n_insts
```

### OpenVera

Not supported.

### Description

Automatically stops the multi-stream scenario generator, when the number of generated transaction descriptors reaches or surpasses the specified value. A value of zero indicates an infinite number of transaction descriptors.

The number of transaction descriptor instances generated by the execution of a multi-stream scenario is the number of transactions reported by the `vmm_ms_scenario::execute()` method, when it returns. Entire scenarios are executed before the generator is stopped, so that the actual number of transaction descriptors generated may be greater than the specified value.

### Example

*Example A-149*

```
`vmm_scenario_gen(atm_cell, "atm trans")

class my_ms_scenario extends vmm_ms_scenario;
   ...
endclass
```

```
program test_ms_scenario;
   ...
   vmm_ms_scenario_gen ms_gen = new("MS Scenario Gen", 10);
   my_ms_scenario ms_scen = new;
   ...
   initial begin
      ...
      ms_gen.stop_after_n_instances = 100;
      ...
   end

endprogram
```

## vmm_ms_scenario_gen::stop_after_n_scenarios

Returns the number of multi-stream scenarios to generate.

### SystemVerilog

```
int unsigned stop_after_n_scenarios
```

### OpenVera

Not supported.

### Description

Automatically stops the multi-stream scenario generator, when the number of generated multi-streams scenarios reaches or surpasses the specified value. A value of zero specifies an infinite number of multi-stream scenarios.

Only the multi-stream scenarios explicitly executed by this instance of the multi-stream scenario generator are counted. Sub-scenarios executed as part of a higher-level multi-stream scenario are not counted.

### Example

*Example A-150*

```
`vmm_scenario_gen(atm_cell, "atm trans")

class my_ms_scenario extends vmm_ms_scenario;
   ...
endclass
program test_ms_scenario;
   ...
   vmm_ms_scenario_gen ms_gen = new("MS Scenario Gen", 10);
```

```
   my_ms_scenario ms_scen = new;
   ...
   initial begin
      ...
      ms_gen.stop_after_n_scenarios = 10;
      ...
   end

endprogram
```

## vmm_ms_scenario_gen::unregister_channel()

Unregisters an output channel.

### SystemVerilog

```
virtual function bit unregister_channel(
     vmm_channel chan)
```

### OpenVera

Not supported.

### Description

Completely unregisters the specified output channel and returns `TRUE`, if it exists in the registry.

### Example

*Example A-151*

```
`vmm_channel(atm_cell)
`vmm_scenario_gen(atm_cell, "atm_trans")

program test_scen;
     vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen",
11);
     atm_cell_channel ms_chan_1=new("MS-CHANNEL-1",
"MY_CHANNEL");
     ...
     initial begin
          vmm_log(log,"Registering channel \n");
          my_ms_gen.register_channel("MS-CHANNEL-
1",ms_chan_1);
          ...
          if(my_ms_gen.unregister_channel(ms_chan_1)
               vmm_log(log,"Channel has been
```

```
unregistered\n");
          ...
     end
endprogram
```

## vmm_ms_scenario_gen::unregister_channel_by_name()

Unregisters an output channel

### SystemVerilog

```
virtual function vmm_channel unregister_channel_by_name(
     string name)
```

### OpenVera

Not supported.

### Description

Unregisters the output channel under the specified name, and returns the unregistered channel. Returns `NULL`, if there is no channel registered under the specified name.

### Example

*Example A-152*

```
`vmm_channel(atm_cell)
`vmm_scenario_gen(atm_cell, "atm_trans")

program test_scen;
     vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen",
11);
     atm_cell_channel ms_chan_1=new("MS-CHANNEL-1",
"MY_CHANNEL");
     atm_cell_channel buffer_chan = new("BUFFER","MY_BC");
     ...
     initial begin
          vmm_log(log,"Registering channel \n");
          my_ms_gen.register_channel("MS-CHANNEL-
1",ms_chan_1);
```

```
          ...
          vmm_log(log,"Unregistered channel by name \n");
          buffer_chan =
my_ms_gen.unregister_channel_by_name("MS-CHANNEL-


1");
          ...
     end
endprogram
```

## vmm_ms_scenario_gen::unregister_ms_scenario()

Unregisters a scenario descriptor.

### SystemVerilog

```
virtual function bit unregister_ms_scenario(
    vmm_ms_scenario scenario)
```

### OpenVera

Not supported.

### Description

Completely unregisters the specified multi-stream scenario descriptor and returns `TRUE`, if it exists in the registry. The unregistered scenario is also removed from the `vmm_ms_scenario_gen::scenario_set[$]` array.

### Example

*Example A-153*

```
class my_ms_scen extends vmm_ms_scenario;
   ...
endclass

program test_scenario;
  vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen", 9);
   my_ms_scen ms_scen = new;
   ...
   initial begin
      my_ms_gen.register_ms_scenario("MS SCEN-1",ms_scen);
      ...
      if(my_ms_gen.unregister_ms_scenario(ms_scen)
         vmm_log(log,"Scenario unregistered \n");
```

```
      else
         vmm_log(log,"Unable to unregister  \n");
      ...
   end
endprogram
```

## vmm_ms_scenario_gen::unregister_ms_scenario_by_name()

Unregisters a scenario descriptor.

### SystemVerilog

```
virtual function vmm_ms_scenario
unregister_ms_scenario_by_name(
     string name)
```

### OpenVera

Not supported.

### Description

Unregisters the multi-stream scenario under the specified name, and returns the unregistered scenario descriptor. Returns `NULL`, if there is no scenario registered under the specified name.

The unregistered scenario descriptor is removed from the `vmm_ms_scenario_gen::scenario_set[$]` array, if it is not also registered under another name.

### Example

*Example A-154*

```
class my_ms_scen extends vmm_ms_scenario;
   ...
endclass

program test_scenario;
  vmm_ms_scenario_gen my_ms_gen = new("MS Scenario Gen", 9);
   my_ms_scen ms_scen = new;
   my_ms_scen buffer_scen =new;
```

```
   ...
   initial begin
      my_ms_gen.register_ms_scenario("MS SCEN-1",ms_scen);
      ...
      buffer_scen =
         my_ms_gen.unregister_ms_scenario_by_name(
            "MY-SCEN-1",ms_scen);
      if(buffer_scen == null)
         vmm_log(log,"Returned null value \n");
     ...
   end
endprogram
```

## vmm_ms_scenario_gen::unregister_ms_scenario_gen()

Unregisters a sub-generator

### SystemVerilog

```
virtual function bit
unregister_ms_scenario_gen(vmm_ms_scenario_gen
scenario_gen)
```

### OpenVera

Not supported.

### Description

Completely unregisters the specified sub-generator and returns `TRUE`, if it exists in the registry.

### Example

*Example A-155*

```
program test_scenario;
   string buffer_ms_gen_name;
   vmm_ms_scenario_gen parent_ms_gen =
      new("Parent-MS-Scen-Gen", 11);
   vmm_ms_scenario_gen child_ms_gen  =
      new("Child-MS-Scen-Gen", 6);
   ...
   initial begin
      vmm_log(log,"Registering sub MS generator \n");
      parent_ms_gen.register_ms_scenario_gen(
         "Child-MS-Gen-1",child_ms_gen);
      ...
      if(parent_ms_gen.unregister_ms_scenario_gen(
         child_ms_gen))
```

```
         vmm_log(log,"Scenario unregistered \n");
      else
         vmm_log(log,"Unable to unregister  \n");
   end

endprogram
```

## vmm_ms_scenario_gen::unregister_ms_scenario_gen_by_name()

Unregisters a sub-generator.

### SystemVerilog

```
virtual function vmm_ms_scenario_gen
     unregister_ms_scenario_gen_by_name(
         string name)
```

### OpenVera

Not supported.

### Description

Unregisters the generator under the specified name, and returns the unregistered generator. Returns `NULL`, if there is no generator registered under the specified name.

### Example

*Example A-156*

```
program test_scenario;
   vmm_ms_scenario_gen parent_ms_gen =
      new("Parent-MS-Scen-Gen", 11);
   vmm_ms_scenario_gen child_ms_gen  =
      new("Child-MS-Scen-Gen", 6);
   vmm_ms_scenario_gen buffer_ms_gen  =
     new("Buffer-MS-Scen-Gen", 6);
   ...
   initial begin
      vmm_log(log,"Registering sub MS generator \n");
      parent_ms_gen.register_ms_scenario_gen(
         "Child-MS-Gen-1",child_ms_gen);
```

```
      parent_ms_gen.register_ms_scenario_gen(
         "Child-MS-Gen-2",child_ms_gen);
      ...
      buffer_ms_gen =
         parent_ms_gen.unregister_ms_scenario_gen_by_name(
            "Child-MS-Gen-1");
   end
endprogram
```

# vmm_notification

This class is used to describe a notification that can be autonomously indicated or reset based on a user-defined behavior, such as the composition of other notifications or external events. Notification descriptors are attached to notifications, using the **`vmm_notify::set_notification()`** method.

## Summary


- vmm_notification::indicate() ..... page A-419
- vmm_notification::reset() ..... page A-421

## vmm_notification::indicate()

Define a method that causes the notification attached to the descriptor to be indicated.

### SystemVerilog

```
virtual task indicate(ref vmm_data status);
```

### OpenVera

Not supported.

### Description

Defines a method that, when it returns, causes the notification attached to the descriptor to be indicated. The value of the **`status`** argument is used as the indicated notification status descriptor. This method is automatically invoked by the notification service interface when a notification descriptor is attached to a notification, using the **`vmm_notify::set_notification()`** method.

This method must be overloaded in a user-defined class extensions. It can be used to implement arbitrary notification mechanisms, such as notifications based on a complex composition of other indications (for example, notification expressions) or external events.

### Example

*Example A-157*

```
class bus_mon extends vmm_xactor;
   static int OBSERVED;
   my_trans tr;
```

```
   function new(...);
    super.new(...);
    this.notify.configure(OBSERVED,vmm_notify::ON_OFF);
   endfunction
   ...
   virtual task main();
   ...
    forever begin
     tr=new();
     ...
     this.notify.indicate(OBSERVED,tr);
     ...
    end
   endtask: main
endclass: bus_mon
```

## vmm_notification::reset()

Defines a method that causes the ON or OFF notification, which is attached to the notification descriptor to be reset.

### SystemVerilog

```
virtual task reset();
```

### OpenVera

Not supported.

### Description

Defines a method that, when it returns, causes the ON or OFF notification, which is attached to the notification descriptor to be reset. This method is automatically invoked by the notification service interface, when a notification definition is attached to a **`vmm_notify::ON_OFF`** notification.

This method must be overloaded in user-defined class extensions.

### Example

*Example A-158*

Example of notification indicated when two other notifications are indicated:

```
class bus_mon extends vmm_xactor;

  static int OBSERVED;
  my_trans tr;
```

```
  function new(...);
  super.new(...);
   this.notify.configure(OBSERVED,vmm_notify::ON_OFF);
  endfunction
  ...
  virtual task main();
  ...
  forever begin
  tr=new();
  ...
  this.notify.indicate(OBSERVED,tr);
  ...
  this.notify.reset();

  end
 endtask: main
endclass: bus_mon
```

# vmm_notify

The **vmm_notify** class implements an interface to the notification service. The notification service provides a synchronization mechanism for concurrent threads or transactors. Unlike **event** variables, the operation of the notification is defined at configuration time. Moreover, notification can include status and timestamp information attached to their indication.

## Summary


- vmm_notify::append_callback() ..... page A-424
- vmm_notify::configure() ..... page A-426
- vmm_notify::copy() ..... page A-428
- vmm_notify::get_notification() ..... page A-429
- vmm_notify::indicate() ..... page A-430
- vmm_notify::is_configured() ..... page A-431
- vmm_notify::is_on() ..... page A-432
- vmm_notify::is_waited_for() ..... page A-433
- vmm_notify::new() ..... page A-435
- vmm_notify::register_vmm_sb_ds() ..... page A-436
- vmm_notify::reset() ..... page A-437
- vmm_notify::set_notification() ..... page A-439
- vmm_notify::status() ..... page A-440
- vmm_notify::terminated() ..... page A-441
- vmm_notify::timestamp() ..... page A-442
- vmm_notify::unregister_callback() ..... page A-443
- vmm_notify::unregister_vmm_sb_ds() ..... page A-445
- vmm_notify::wait_for() ..... page A-446
- vmm_notify::wait_for_off() ..... page A-447
- vmm_notify_callbacks::indicated() ..... page A-449
- `vmm_notify_observer ..... page A-453
- vmm_notify_observer::new() ..... page A-454

## vmm_notify::append_callback()

Registers a callback extension.

### SystemVerilog

```
function void append_callback(int
notification_id,
     vmm_notify_callbacks cbs);
```

### OpenVera

```
task append_callback(integer event_id,
     rvm_notify_callbacks cbs);
```

### Description

Appends the specified callback extension to the list of registered callbacks, for the specified notification. All registered callback extensions are invoked, when the specified notification is indicated.

### Example

*Example A-159*

```
class my_callbacks extends vmm_notify_callbacks;
   virtual function void indicated(vmm_data status);
      ...
   endfunction
endclass

program vmm_notify_test;
   initial begin
      int EVENT_A = 1;
     vmm_log log = new("Notify event", "vmm_notify_test");
      vmm_notify notify = new(log);
      my_callbacks my_callbacks_inst = new;
      void'(notify.configure(EVENT_A));
```

```
      ...
      `vmm_note(log, "Appending vmm notify call back");
      notify.append_callback(EVENT_A,my_callbacks_inst);
      ...
   end
endprogram
```

## vmm_notify::configure()

Defines a new notification.

### SystemVerilog

```
virtual function int
           configure(int notification_id = -1,
     sync_e sync = ONE_SHOT);
```

### OpenVera

Not supported.

### Description

Defines a new notification associated with the specified unique identifier. If a negative identifier value is specified, a new, unique identifier greater than 1,000,000 is returned. The thread synchronization mode of a notification is defined when the notification is configured, and not when it is triggered or waited upon, using one of the **`vmm_notify::ONE_SHOT`**, **`vmm_notify::BLAST`**, or **`vmm_notify::ON_OFF`** synchronization types. This definition timing prevents a notification from being misused by the triggering or waiting threads.

*Table A-12 Notification Synchronization Mode Enumerated Values*

*Table A-13*

| Enumerated Value | Broadcasting Operation |
|---|---|
| vmm_notify::ONE_SHOT | Only threads currently waiting for the notification to be indicated are notified. |
| vmm_notify::BLAST | All threads waiting for the notification to be indicated in the same timestep at the indication are notified. This mode eliminates certain types of race conditions. |
| vmm_notify::ON_OFF | The notification is level-sensitive. Notifications remain notified until explicitly reset. Threads waiting for a notification that is still notified will not wait. This mode eliminates certain types of race conditions. |

A warning may be generated, if a notification is configured more than once.

Notification identifiers numbered from 1,000,000 and above are reserved for automatically generated notification identifiers. Predefined notification identifiers in the VMM base classes use identifiers from 999,999 and below. User-defined notification identifiers can thus use values from 0 and above.

## vmm_notify::copy()

Copies the current configuration of this notification service interface.

### SystemVerilog

```
virtual function vmm_notify copy(vmm_notify to = null);
```

### OpenVera

Not supported.

### Description

Copies the current configuration of this notification service interface to the specified instance. If no instance is specified, a new instance is allocated using the same message service interface as the original one. A reference to the copied target instance is returned.

Only the notification configuration information is copied and merged with any pre-configured notification in the destination instance. Copied notification configuration replaces any pre-existing configuration for the same notification identifier. Status and timestamp information is **not** copied.

## vmm_notify::get_notification()

Gets the notification descriptor associated with the notification.

### SystemVerilog

```
virtual function vmm_notification
          get_notification(int notification_id);
```

### OpenVera

Not supported.

### Description

Gets the notification descriptor associated with the specified notification, if any. If no notification descriptor is associated with the specified notification, then it returns *null*.

## vmm_notify::indicate()

Indicates the specified notification with the optional status descriptor.

### SystemVerilog

```
virtual function void indicate(int notification_id,
     vmm_data status = null);
```

### OpenVera

Not supported.

### Example

*Example A-160*

```
class consumer extends vmm_xactor;
   ...
   virtual task main();
      ...
      forever begin
         ...
         this.in_chan.get(tr);
         tr.notify.indicate(vmm_data::STARTED);
         ...
      end
   endtask: main
endclass: consumer
```

## vmm_notify::is_configured()

Checks whether the specified notification is configured or not.

### SystemVerilog

```
virtual function int is_configured(int notification_id);
```

### OpenVera

Not supported.

### Description

Checks whether the specified notification is currently configured or not. If this method returns 0, then the notification is not configured. Otherwise, it returns an integer value corresponding to the current **`vmm_notify::ONE_SHOT`**, **`vmm_notify::BLAST,`** or **`vmm_notify::ON_OFF`** configuration.

## vmm_notify::is_on()

Check whether the specified **`vmm_notify::ON_OFF`** notification is currently in the **`notify`** state or not.

### SystemVerilog

```
virtual function bit is_on(int notification_id);
```

### OpenVera

Not supported.

### Description

If this method returns TRUE, then the notification is in the notify state, and any call to the **`vmm_notify::wait_for()`** method will not block. A warning is generated, if this method is called on any other types of notifications.

## vmm_notify::is_waited_for()

Checks whether a thread is currently waiting for the specified notification or not.

### SystemVerilog

```
virtual function bit is_waited_for(int notification_id);
```

### OpenVera

Not supported.

### Description

Checks whether a thread is currently waiting for the specified notification or not, including waiting for an ON or OFF notification to be reset. It is an error to specify an unconfigured notification. The function returns TRUE, if there is a thread known to be waiting for the specified notification.

Note that the knowledge about the number of threads waiting for a particular notification is not definitive, and may be out of date. As threads call the **`vmm_notify::wait_for()`** method, the fact that they are waiting for the notification is recorded. Once the notification is indicated and each thread returns from the method call, the fact that they are no longer waiting is also recorded. However, if the threads are externally terminated through the **`disable`** statement or a timeout, the fact that they are no longer waiting cannot be recorded. In this case, it is up to the terminated threads to report that they are no longer waiting, by calling the **`vmm_notify::terminated()`** method.

When a notification is reset with a hard reset, no threads are assumed to be waiting for any notification.

## vmm_notify::new()

Creates a new instance of this class.

### SystemVerilog

```
function new(vmm_log log);
```

### OpenVera

Not supported.

### Description

Creates a new instance of this class, using the specified message service interface to generate an error and debug messages.

## vmm_notify::register_vmm_sb_ds()

For more information on this class, refer to the *VMM Scoreboard User Guide*.

## vmm_notify::reset()

Resets the specified notification.

### SystemVerilog

```
virtual function void reset(int notification_id = -1,
    reset_e rst_typ = SOFT);
```

### OpenVera

Not supported.

### Description

A **vmm_notify::SOFT** reset clears the specified **ON_OFF** notification, and restarts the **vmm_notification::indicate()** and **vmm_notification::reset()** methods on any attached notification descriptor. A **vmm_notify::HARD** reset clears all status information and attached notification descriptor on the specified event, and further assumes that no threads are waiting for that notification. If no notification is specified, all notifications are reset.

### Example

*Example A-161*

The following example shows definitions of three user-defined notifications:

```
class bus_mon extends vmm_xactor;
   static int EVENT_A = 0;
   static int EVENT_B = 1;
   static int EVENT_C = 2;
```

```
   function new(...);
      super.new(...);
      super.notify.configure(this.EVENT_A);
      super.notify.configure(this.EVENT_B,
                             vmm_notify::ON_OFF);
      super.notify.configure(this.EVENT_C,
                             vmm_notify::BLAST);
   endfunction
   ...
   virtual task main();
     ...
     forever begin
      ...
      super.notify.indicate(this.EVENT_A);
      ...
      super.notify.reset(this.EVENT_A);
      ...
     end
   endtask: main
endclass: bus_mon
```

## vmm_notify::set_notification()

Defines the notification, using the notification descriptor.

### SystemVerilog

```
virtual function void
      set_notification(int notification_id,
     vmm_notification ntfy = null);
```

### OpenVera

Not supported.

### Description

Defines the specified notification, using the specified notification descriptor. If the descriptor is *null*, then the notification is undefined and can only be indicated using the **vmm_notify::indicate()** method. If a notification is already defined, the new definition replaces the previous definition.

## vmm_notify::status()

Returns the status descriptor that is associated with the notification.

### SystemVerilog

```
virtual function vmm_data status(int notification_id);
```

### OpenVera

Not supported.

### Description

Returns the status descriptor that is associated with the specified notification, when it was last indicated. It is an error to specify an unconfigured notification.

## vmm_notify::terminated()

Indicates that a thread waiting for the specified notification is disabled.

### SystemVerilog

```
virtual function void terminated(int notification_id);
```

### OpenVera

Not supported.

### Description

Indicates to the notification service interface that a thread waiting for the specified notification is disabled, and is no longer waiting.

## vmm_notify::timestamp()

Returns the simulation time when the notification was last indicated.

### SystemVerilog

```
virtual function time timestamp(int notification_id);
```

### OpenVera

Not supported.

### Description

Returns the simulation time when the specified notification was last indicated. It is an error to specify an unconfigured notification.

## vmm_notify::unregister_callback()

Unregisters a callback extension.

### SystemVerilog

```
function void unregister_callback(
    int                  notification_id,
    vmm_notify_callbacks sb);
```

### OpenVera

```
task unregister_callback(integer              event_id,
    rvm_notify_callbacks sb);
```

### Description

Unregisters the specified callback extension from the notification service interface, for the specified notification. An error is generated, if the specified callback extension was not previously registered with the specified notification.

### Example

*Example A-162*

```
class my_callbacks extends vmm_notify_callbacks;
   virtual function void indicated(vmm_data status);
      ...
   endfunction
endclass

program vmm_notify_test;
   initial begin
      int EVENT_A = 1;
     vmm_log log = new("Notify event", "vmm_notify_test");
      vmm_notify notify = new(log);
```

```
      my_callbacks my_callbacks_inst = new;
      void'(notify.configure(EVENT_A));
      ...
      `vmm_note(log, "Unregistering vmm notify call back");
     notify.unregister_callback(EVENT_A,my_callbacks_inst);
      ...
   end
endprogram
```

## vmm_notify::unregister_vmm_sb_ds()

For more information on this class, refer to the *VMM Scoreboard User Guide*.

## vmm_notify::wait_for()

Suspends the execution thread, until the specified notification is notified.

### SystemVerilog

```
virtual task wait_for(int notification_id);
```

### OpenVera

Not supported.

### Description

It is an error to specify an unconfigured notification. Use the **`vmm_notify::status()`** function to retrieve any status descriptor attached to the indicated notification.

### Example

*Example A-163*

```
class consumer extends vmm_xactor;
   ...
   virtual task main();
      ...
      while (1) begin
         ...
         this.in_chan.peek(tr);
         tr.notify.wait_for(vmm_data::ENDED);
         this.in_chan.get(tr);
         ...
      end
   endtask: main
endclass: consumer
```

## vmm_notify::wait_for_off()

Suspends the execution thread, until the specified **`vmm_notify::ON_OFF`** notification is reset.

### SystemVerilog

```
virtual task wait_for_off(int notification_id);
```

### OpenVera

Not supported.

### Description

It is an error to specify an unconfigured or a non-ON or OFF notification. The status returned by subsequent calls to the **`vmm_notify::status()`** function is undefined.

# vmm_notify_callbacks

Facade class for callback methods provided by the notification service. User-defined extensions of this class must be registered with specific instances of the notification service interface and for specific notifications, using the `vmm_notify::append_callback()` method.

This class is a virtual class and cannot be instantiated on its own.

## Summary


- `vmm_notify_callbacks::indicated()` ..... page A-449

## vmm_notify_callbacks::indicated()

Reports that a notification is indicated.

### SystemVerilog

```
virtual function void indicated(vmm_data status);
```

### OpenVera

```
virtual task indicated(rvm_data status);
```

### Description

This method is invoked whenever the notification corresponding to the callback extension is indicated. The status is a reference to the status descriptor, which is specified to the **`vmm_notify::indicate()`** method that caused the notification to be indicated.

The purpose of this callback is similar to the **`vmm_notify::wait_for()`** method. However, unlike the **`vmm_notify:;wait_for()`** method, it reliably reports multiple indications of the same notification during the same timestep.

### Example

```
class my_callbacks extends vmm_notify_callbacks;

  virtual function void indicated(vmm_data status);
     ...
  endfunction

endclass
```

```
class bus_mon extends vmm_xactor;
    static int OBSERVED;
     my_trans tr;

      function new(...);
         super.new(...);
       this.notify.configure(OBSERVED,vmm_notify::ON_OFF);
       endfunction
       ...
     virtual task main();
          ...
         forever begin
            tr=new();
             ...
            this.notify.indicate(OBSERVED,tr);
             ...
         end
     endtask: main
endclass: bus_mon

class env extends vmm_env;
    my_callbacks my_callbacks_inst = new();
    bus_mon mon=new();

    function void build();
       ...
      `vmm_note(log, "Appending vmm notify call back");
       mon.notify.append_callback (bus_mon::OBSERVED,
my_callbacks_inst);
       ...
    endfunction : build

endclass
```

# vmm_notify_observer#(T,D)

Simplifies subscription to a notification callback method.

## SystemVerilog

```
class vmm_notify_observer #(type T, type D = vmm_data)
   extends vmm_notify_callbacks
```

## Description

The `vmm_notify_observer` class is a parameterized extension of `vmm_notify_callbacks`. Any subscriber (scoreboard, coverage model, and so on) can get the transaction status whenever a notification event is indicated. The `` `vmm_notify_observer `` macro is provided to specify the observer and its method name to be called.

## Example

```
class scoreboard;
   virtual function void observe_trans(ahb_trans tr);
      ...
   endfunction
endclass
`vmm_notify_observer(scoreboard, observe_trans)
```

Instantiate parameterized `vmm_notify_observer`, passing the subscriber handle, `vmm_notify` handle and the notification ID.

```
scoreboard  sb = new();
vmm_notify_observer#(scoreboard, ahb_trans)
   observe_start = new(sb, mon.notify, mon.TRANS_START);
```

## Summary


- `vmm_notify_observer ..... page A-453
- vmm_notify_observer::new() ..... page A-454

## `vmm_notify_observer

Defines a parameterized class in the style of the `vmm_notify_observer` class.

### SystemVerilog

```
`define vmm_notify_observer(classname, methodname)
```

### Description

Defines a parameterized class in the style of the `vmm_notify_observer` class, with the specified name, and calling the specified `T::methodname(D.status)` method. Useful for defining a subscription class for an observer with a different observation method.

### Example

```
class scoreboard;
   virtual function void observe_trans(ahb_trans tr);
      ...
   endfunction
endclass
`vmm_notify_observer(scoreboard, observe_trans)
```

## vmm_notify_observer::new()

Appends a callback method to invoke the `T::observe(D.status)` method in the specified instance.

### SystemVerilog

```
function vmm_notify_observer::new(T observer,
   vmm_notify ntfy, int notification_id)
```

### Description

Appends a callback method to invoke the `T::observe(D)` method in the specified instance, whenever the specified indication is notified on the specified Notification Service Interface.

### Example

```
vmm_notify_observer#(scoreboard, ahb_trans)
   observe_start = new(sb, mon.notify, mon.TRANS_START);
```

# vmm_object

The `vmm_object` class is a virtual class that is used as the common base class for all VMM related classes. This helps to provide parent or child relationships for class instances. Additionally, it provides local, relative, and absolute hierarchical naming.

## Summary


- vmm_object::create_namespace() ..... page A-456
- vmm_object::display() ..... page A-457
- vmm_object::find_child_by_name() ..... page A-458
- vmm_object::find_object_by_name() ..... page A-459
- vmm_object::get_hier_inst_name() ..... page A-460
- vmm_object::get_log() ..... page A-461
- vmm_object::get_namespaces() ..... page A-462
- vmm_object::get_num_children() ..... page A-463
- vmm_object::get_num_roots() ..... page A-464
- vmm_object::get_nth_child() ..... page A-465
- vmm_object::get_nth_root() ..... page A-466
- vmm_object::get_object_hiername() ..... page A-467
- vmm_object::get_object_name() ..... page A-468
- vmm_object::get_parent() ..... page A-469
- vmm_object::get_parent_object() ..... page A-470
- vmm_object::get_root_object() ..... page A-471
- vmm_object::get_type() ..... page A-472
- vmm_object::get_typename() ..... page A-473
- vmm_object::implicit_phasing() ..... page A-474
- vmm_object::is_implicitly_phased() ..... page A-475
- vmm_object::is_parent_of() ..... page A-476
- vmm_object::kill_object() ..... page A-477
- vmm_object::new() ..... page A-478
- vmm_object::print_hierarchy() ..... page A-479
- vmm_object::psdisplay() ..... page A-481
- vmm_object::set_object_name() ..... page A-482
- vmm_object::set_parent() ..... page A-483
- vmm_object::set_parent_object() ..... page A-485
- vmm_object::type_e ..... page A-486
- `foreach_vmm_object() ..... page A-488
- `foreach_vmm_object_in_namespace() ..... page A-489

## vmm_object::create_namespace()

Defines a namespace with specified default object inclusion policy.

### SystemVerilog

```
function bit create_namespace(string name, namespace_typ_e
typ = OUT_BY_DEFAULT);
```

### Description

Defines a namespace with the specified default object inclusion policy. A namespace must be previously created using this method, before it can be used or referenced. Returns true, if the namespace was successfully created. The empty name space ("") is reserved and cannot be defined.

### Example

```
class A extends vmm_object;
   function new (string name, vmm_object parent=null);
      super.new (parent, name);
      vmm_object::create_namesapce("NS1",
         vmm_object::IN_BY_DEFAULT);
   endfunction
endclass
```

## vmm_object::display()

Displays a description of the object to the standard output.

### SystemVerilog

```
virtual function void display(string prefix = "");
```

### OpenVera

Not supported.

### Description

Displays the image returned by "vmm_object::type_e" to the standard output. Each line of the output will be prefixed with the specified argument `prefix`.

If this method conflicts with a previously declared method in a class, which is now based on the `vmm_object` class, it can be removed by defining the `'VMM_OBJECT_NO_DISPLAY` symbol at compile-time.

### Example

*Example A-164*

```
class trans_data extends vmm_data;
   byte data;
   ...
endclass

initial begin
   trans_data trans;
   trans.display("Test Trans: ");
end
```

## vmm_object::find_child_by_name()

Finds the named object relative to this object.

### SystemVerilog

```
function vmm_object vmm_object::find_child_by_name(
   string name, string space = "");
```

### Description

Finds the named object, interpreting the name as a hierarchical name relative to this object in the specified namespace. If the name is a match pattern or regular expression, the first object matching the name is returned. Returns null, if no child was found under the specified name.

### Example

```
class D extends vmm_object;
   ...
endclass
class E extends vmm_object;
   D d1;
   function new(string name, vmm_object parent=null);
      ...
      d1 = new ("d1",this);
   endfunction
endclass
...
initial begin
   vmm_object obj;
   E e1= new ("e1");
   ...
   obj = e1.find_child_by_name ("d1");
   ...
end
```

## vmm_object::find_object_by_name()

Finds the named object in the specified namespace.

### SystemVerilog

```
static function vmm_object
vmm_object::find_object_by_name(string name,
   string space = "");
```

### Description

Finds the named object, interpreting the name as an absolute name in the specified namespace. If the name is a match pattern or regular expression, the first object matching the name is returned.

Returns null, if no object was found under the specified name.

### Example

```
class D extends vmm_object;
   ...
endclass
class E extends vmm_object;
   D d1;
   function new(string name, vmm_object parent=null);
      ...
      d1 = new ("d1");
   endfunction
endclass
...
initial begin
   vmm_object obj;
   ...
   obj = E :: find_object_by_name ("d1");
   ...
end
```

## vmm_object::get_hier_inst_name()

Returns the hierarchical instance name of the object.

### SystemVerilog

```
function string get_hier_inst_name();
```

### OpenVera

Not supported.

### Description

Returns the hierarchical instance name of the object. The instance name is composed of the dot-separated instance names of the message service interface of all the parents of the object.

The hierarchical name is returned, whether or not the message services interfaces are using hierarchical or flat names.

### Example

*Example A-165*

```
class tb_env extends vmm_env;
   tr_scenario_gen gen1;
   ...
endclass
initial begin
   string str;
   tb_env env;
   ...
   str = env.s1.gen1.get_hier_inst_name();
   `vmm_note(log, str);
end
```

## vmm_object::get_log()

Returns the `vmm_log` instance of this object.

### SystemVerilog

```
virtual function vmm_log vmm_object::get_log();
```

### Description

Returns the `vmm_log` instance of this object, or the nearest enclosing object. If no `vmm_log` instance is available in the object genealogy, a default global `vmm_log` instance is returned.

### Example

```
class ABC extends vmm_object;
   vmm_log log = new("ABC", "class");
   ...
   function vmm_log get_log();
      return this.log;
   endfunction
...
endclass

vmm_log test_log;
ABC abc_inst = new("test_abc");
initial begin
   test_log = abc_inst.get_log();
   ...
end
```

## vmm_object::get_namespaces()

Returns all namespaces created by the `create_namespace()` method.

### SystemVerilog

```
function void get_namespaces(output string names[]);
```

### Description

This method returns all namespaces created by the `create_namespace() method` that belong to a dynamic array of strings as specified by `names[]`.

### Example

```
initial begin
   string ns_array[];
   ...
   vmm_object::get_namespaces(ns_array);
   ...
end
```

## vmm_object::get_num_children()

Gets the total number of children for this object.

### SystemVerilog

```
function int vmm_object::get_num_children();
```

### Description

Gets the total number of children object for this object.

### Example

```
class C extends vmm_object;
   ...
endclass
class D extends vmm_object;
   ...
endclass
class E extends vmm_object;
   C c1;
   D d1;
   D d2;
   function new(string name, vmm_object parent=null);
      ...
      c1 = new ("c1",this);
      d1 = new ("d1");
      d2 = new ("d2",this);
   endfunction
endclass
int num_children;
   initial begin
   E e1 = new ("e1");
   ...
   num_children = e1.get_num_children;
   ...
end
```

## vmm_object::get_num_roots()

Gets the total number of root objects in the specified namespace.

### SystemVerilog

```
static function int vmm_object::get_num_roots(
   string space = "");
```

### Description

Gets the total number of root objects in the specified namespace.

### Example

```
class D extends vmm_object;
   ...
endclass
class E extends vmm_object;
   D d1;
   D d2;
   function new(string name, vmm_object parent=null);
      ...
      d1 = new ("d1");
      d2 = new ("d2");
   endfunction
endclass
...
int num_roots;
initial begin
   E e1 = new ("e1");
   ...
   num_roots = E :: get_num_roots(); //Returns 2
   ...
end
```

## vmm_object::get_nth_child()

Returns the *n*th child of this object.

### SystemVerilog

```
function vmm_object vmm_object::get_nth_child(int n);
```

### Description

Returns the *n*th child of this object. Returns null, if there is no child.

### Example

```
class C extends vmm_object;
   ...
endclass
class D extends vmm_object;
   ...
endclass
class E extends vmm_object;
   C c1;
   D d1;
   D d2;
   function new(string name, vmm_object parent=null);
      c1 = new ("c1",this);
      d1 = new ("d1");
      d2 = new ("d2",this);
   endfunction
endclass
initial begin
   vmm_object obj;
   string name;
   E e1 = new ("e1");
   obj = e1.get_nth_child(0);
   name = obj.get_object_name();  //Returns c1
   ...
end
```

## vmm_object::get_nth_root()

Returns the *n*th root object in the specified namespace.

### SystemVerilog

```
static function vmm_object vmm_object::get_nth_root(int n,
   string space = "");
```

### Description

Returns the *n*th root object in the specified namespace. Returns null, if there is no such root.

### Example

```
class D extends vmm_object;
   ...
endclass
class E extends vmm_object;
   D d1;
   D d2;
   function new(string name, vmm_object parent=null);
      ...
      d1 = new ("d1");
      d2 = new ("d2");
   endfunction
endclass
...
int num_roots;
initial begin
   vmm_object root;
   E e1 = new ("e1");
   ...
   root= E :: get_nth_root(0); //Returns d1
   ...
end
```

## vmm_object::get_object_hiername()

Gets the complete hierarchical name of this object.

### SystemVerilog

```
function string vmm_object::get_object_hiername(
   vmm_object root = null, string space = "");
```

### Description

Gets the complete hierarchical name of this object in the specified namespace, relative to the specified root object. If no root object is specified, returns the complete hierarchical name of the object. The instance name is composed of the period-separated instance names of the message service interface of all the parents of the object.

### Example

```
class D extends vmm_object;
   ...
endclass
class E extends vmm_object;
   D d1;
   function new(string name, vmm_object parent=null);
      ...
      d1 = new ("d1",this);
   endfunction
endclass
...
initial begin
   string hier_name;
   E e1 = new ("e1");
   ...
   hier_name = e1.d1.get_object_hiername();
   ...
end
```

## vmm_object::get_object_name()

Gets the local name of this object

### SystemVerilog

```
function string vmm_object::get_object_name(
   string space = "");
```

### Description

Gets the local name of this object, in the specified namespace. If no namespace is specified, then returns the actual name of the object.

### Example

```
class C extends vmm_object;
   function new(string name, vmm_object parent=null);
      super.new (parent,name);
   endfunction
endclass
...
initial begin
   string obj_name;
   C c1 = new ("c1");
   ...
   obj_name = c1.get_object_name(); //Returns c1
   ...
end
```

## vmm_object::get_parent()

Returns a parent object.

### SystemVerilog

```
function vmm_object get_parent(
     vmm_object::type_e typ = VMM_OBJECT);
```

### OpenVera

Not supported.

### Description

Returns the parent object of the specified type, if any. Returns `NULL`, if no such parent is found. Specifying `VMM_OBJECT` returns the immediate parent of any type.

### Example

*Example A-166*

```
class tb_env extends vmm_env;
   tr_scenario_gen gen1;
   function new(string inst, vmm_consensus end_vote);
      gen1.set_parent_object(this);
   endfunction
endclass
initial begin
   tb_env env;
   if (env.gen1.randomized_obj.get_parent() != env.gen1)
begin
    `vmm_error(log, "Factory instance in atomic_gen returns
wrong parent");
   end
end
```

## vmm_object::get_parent_object()

Returns the parent of this object.

### SystemVerilog

```
function vmm_object vmm_object::get_parent_object(string
space = "");
```

### Description

Returns the parent object of this object for specified namespace, if any. Returns null, if no parent is found. A root object contains no parent.

### Example

```
class C extends vmm_object;
   ...
endclass
class D extends vmm_object;
   C c1;
   function new(string name, vmm_object parent=null);
   c1 = new ("c1",this);
   endfunction
endclass

initial begin
   vmm_object parent;
   D d1 = new ("d1");
   parent = d1.c1.get_parent_object;
end
```

## vmm_object::get_root_object()

Gets the root parent of this object.

### SystemVerilog

```
function vmm_object vmm_object::get_root_object(
   string space = "");
```

### Description

Gets the root parent of this object, for the specified namespace.

### Example

```
class C extends vmm_object;
   ...
endclass
class D extends vmm_object;
   C c1;
   function new(string name, vmm_object parent=null);
      c1 = new ("c1",this);
   endfunction
endclass
class E extends vmm_object;
   D d1;
   function new(string name, vmm_object parent=null);
      ...
      d1 = new ("d1",this);
   endfunction
endclass
...
initial begin
   vmm_object root;
   E e1 = new ("e1");
   root = e1.d1.c1.get_root_object;
   ...
end
```

## vmm_object::get_type()

Returns the type of the object.

### SystemVerilog

```
function vmm_object::type_e get_type();
```

### OpenVera

Not supported.

### Description

Returns the type of this `vmm_object` extension.

Returns the `VMM_OBJECT`, if it is not one of the known VMM class extensions. `VMM_UNKNOWN` is purely an internal value, and is never returned.

### Example

*Example A-167*

```
class tb_env extends vmm_env;
   tr_scenario_gen gen1;
   gen1.set_parent_object(this);
endclass

initial begin
   tb_env env;
   if (env.get_type() != vmm_object::VMM_ENV)
      begin
         `vmm_error(log, "Wrong type returned from vmm_env
         instance");
      end
end
```

## vmm_object::get_typename()

Returns the name of the actual type of this object.

### SystemVerilog

```
pure virtual function string vmm_object::get_typename();
```

### Description

This function is implemented in the `vmm_typename(string name) macro. It returns the type of this vmm_object extension. However, it will not return an appropriate vmm_object if `vmm_typename(name) is not used in the corresponding class.

### Example

```
class ahb_gen extends vmm_group;
   `vmm_typename (ahb_gen)
      function new (string name);
       super.new (get_typename(), name);
   endfunction
endclass
```

## vmm_object::implicit_phasing()

If the `is_on` argument is false, inhibits the implicit phasing for this object and all of its children objects.

### SystemVerilog

```
virtual function void vmm_object::implicit_phasing(
   bit is_on);
```

### Description

If the `is_on` argument is false, inhibits the implicit phasing for this object and all of its children objects. Used to prevent a large object hierarchy that does not require phasing from being needlessly walked by the implicit phaser (for example, a RAL model). By default, implicit phasing is enabled.

### Example

```
class subsys_env extends vmm_subenv;
   ...
endclass

class sys_env extends vmm_subenv;
   subsys_env subenv1;
   ...
   function void build();
      ...
      subenv1 = new ("subenv1", "subenv1");
      subenv1.set_parent_object(this);
      subenv1.implicit_phasing(0);
      ...
   endfunction
   ...
endclass
```

## vmm_object::is_implicitly_phased()

Returns true, if the implicit phasing is enabled for this object.

### SystemVerilog

```
virtual function bit vmm_object::is_implicitly_phased();
```

### Description

Returns true, if the implicit phasing is enabled for this object.

### Example

```
class subsys_env extends vmm_subenv;
   ...
endclass

class sys_env extends vmm_env;
   subsys_env subenv1;
   ...
   function void build();
      ...
      subenv1 = new ("subenv1", "subenv1");
      subenv1.set_parent_object(this);
      subenv1.implicit_phasing(0);
      if(subenv1.is_implicitly_phased)
         `vmm_error(log, "Implict Phasing for subenv1 not
           disabled");
      ...
   endfunction
   ...
endclass
```

## vmm_object::is_parent_of()

Returns true, if the specified object is a parent of this object.

### SystemVerilog

```
function bit vmm_object::is_parent_of(vmm_object obj,
   string space ="");
```

### Description

Returns true, if the specified object is a parent of this object under specified argument `space` namespace.

### Example

```
class sub extends vmm_subenv;
   ...
endclass

class tb_env extends vmm_env;
   sub s1 ;
   ...
   virtual function void build();
      super.build();
      s1 = new ("s1");
      s1.set_parent_object(this);
      if (!this.is_parent_of(s1))
         `vmm_error(log, "Unable to set parent for s1");
      ...
   endfunction
endclass
```

## vmm_object::kill_object()

Clears cross-references to this object and its children.

### SystemVerilog

```
Virtual function void vmm_object::kill_object();
```

### Description

Clears cross-references to this object and all its children, so that the entire object hierarchy rooted at the object can be garbage collected. Killing the root object enables the garbage collection of the entire object hierarchy underneath it, unless there are other references to an object within that hierarchy. Any external reference to any object in a hierarchy, prevents the garbage collection of that object.

### Example

```
class C extends vmm_object;
   function new(string name, vmm_object parent=null);
      super.new (parent,name);
   endfunction
endclass
class D extends vmm_object;
   C c1;
   function new(string name, vmm_object parent=null);
      super.new (parent,name);
      c1 = new ("c1",this);
   endfunction
endclass
initial begin
   D d1 = new ("d1");
   d1.kill_object;
end
```

## vmm_object::new()

Constructs a new instance of this object.

### SystemVerilog

```
function void vmm_object::new(vmm_object parent = null,
   string name = "[Anonymous]", bit disable_hier_insert = 0);
```

### Description

Constructs a new instance of this object, optionally specifying another object as its parent. The specified name cannot contain any colons (:). Specified argument `disable_hier_insert` indicates whether hierarchical insertion needs to be enabled or not.

To add an object to the parent-child hierarchical structure, set `disable_hier_insert` argument to 1.

### Example

```
class A extends vmm_object;
   function new (string name, vmm_object parent=null);
      super.new (parent, name);
   endfunction
endclass
```

## vmm_object::print_hierarchy()

Prints the object hierarchy.

### SystemVerilog

```
function void  print_hierarchy( vmm_object root = null, bit
verbose=0);
```

### Description

Prints the object hierarchy that is rooted at the specified object. Prints the hierarchy for all roots, if no root is specified.

This method shows the desired object hierarchy, when you ensure that the parent-child relationship is created across different components, either at instantiation time or through `vmm_object::set_parent_object`. `verbose` could be passed as 1 to enabled the verbose option while displaying.

### Example

```
class D extends vmm_object;
   ...
endclass
class E extends vmm_object;
   D d1;
   function new(string name, vmm_object parent=null);
      ...
      d1 = new ("d1",this);
   endfunction
endclass
initial begin
   E e1 = new ("e1");
   ...
   E :: print_hierarchy();
   ...
```

```
end
```

## vmm_object::psdisplay()

Creates a description of the object.

### SystemVerilog

```
virtual function string vmm_object::psdisplay(
   string prefix = "");
```

### Description

Creates a human-readable description of the content of this object. Each line of the image is prefixed with the specified prefix.

### Example

```
class D extends vmm_object;
   ...
   function string psdisplay(string prefix = "");
      ...
   endfuntion
endclass
...
vmm_log log = new ("Test", "main");
initial begin
   D d1 = new ("d1");
   ...
   `vmm_note (log, d1.psdisplay);
   ...
end
```

## vmm_object::set_object_name()

Sets or replaces the name of this object in the specified namespace.

### SystemVerilog

```
function void vmm_object::set_object_name(string name,
   string space = "");
```

### Description

This method is used to set or replace the name of this object in the specified namespace. If no namespace is specified, the name of the object is replaced. If a name is not specified for a namespace, it defaults to the object name. Names in a named namespace may contain colons (:) to create additional levels of hierarchy, or may be empty to skip a level of hierarchy. A name starting with a caret (^) indicates that it is a root in the specified namespace. However, this does not apply to the object name where parentless objects create roots in the default namespace.

### Example

```
class E extends vmm_object;
   ...
endclass
initial begin
   vmm_object obj;
   E e1 = new ("e1");
   vmm_object::create_namespace("NS1",
      vmm_object::IN_BY_DEFAULT);
   ...
   obj = e1;
   obj.set_object_name ("new_e1","NS1");
   ...
end
```

## vmm_object::set_parent()

Specifies a parent object.

### SystemVerilog

```
function void set_parent(vmm_object parent);
```

### OpenVera

Not supported.

### Description

Specifies a new parent object to this object. Specifying a `NULL` parent breaks any current parent or child relationship. An object may contain only one parent, but the identity of a parent can be changed dynamically.

If this object and the parent object are known to contain their own instance of the message service interface, then the `vmm_log` instance in the parent is specified as being above the `vmm_log` instance in the child by calling `parent.is_above(this)`. The instance names of the message service interfaces can then be subsequently made hierarchical by using the "vmm_log::use_hier_inst_name()" method.

The presence of the `vmm_object` base class being optional, it is not possible to call this method in code designed to be reusable with and without this base class. To that effect, the `'VMM_OBJECT_SET_PARENT(_parent, _child)` macro should be used instead. This macro calls this method, if the `vmm_object` base class is present, but do nothing if not.

## Examples

### *Example A-168*

```
this.notify = new(this.log);
this.notify.set_parent_object(this);
```

### *Example A-169*

```
this.notify = new(this.log);
`VMM_OBJECT_SET_PARENT(this.notify, this)
```

## vmm_object::set_parent_object()

Sets or replaces the parent of this object.

### SystemVerilog

```
function void vmm_object::set_parent_object(
   vmm_object parent);
```

### Description

Specifies a new parent object to this object. Specifying a null parent, breaks any current parent or child relationship. An object may contain only one parent, but the identity of a parent can be changed dynamically.

### Example

```
class C extends vmm_object;
   function new(string name, vmm_object parent=null);
      super.new (parent,name);
   endfunction
endclass
class D extends vmm_object;
   C c1;
   function new(string name, vmm_object  parent=null);
      super.new (parent,name);
      c1 = new ("c1",this);
   endfunction
endclass

initial begin
   D d1 = new ("d1");
   D d2 = new ("d2");
   d1.c1.set_parent_object (d2);
end
```

## vmm_object::type_e

Returns the type of this object.

### SystemVerilog

```
typedef enum {
     VMM_UNKNOWN, VMM_OBJECT, VMM_DATA, VMM_SCENARIO,
     VMM_MS_SCENARIO, VMM_CHANNEL, VMM_NOTIFY, VMM_XACTOR,
     VMM_SUBENV, VMM_ENV, VMM_CONSENSUS, VMM_TEST
} type_e
```

### OpenVera

Not supported.

### Description

Value returned by the "vmm_object::type_e" method to identify the type of this vmm_object extension. Once the type is known, a reference to a vmm_object can be cast into the corresponding class type.

The VMM_UNKNOWN type is an internal value, and never returned by the "vmm_object::type_e" method.

The VMM_OBJECT is returned when the type of the object cannot be determined, or to specify any object type to the "vmm_object::type_e" method.

### Example

*Example A-170*

```
program test;
   class tb_env extends vmm_env;
```

```
      type_e env_c_type;
      function new();
         super.new("tb_env");
         end_vote.set_parent_object(this);
         env_c_type = get_type();
      endfunction
   endclass
   initial
   begin
      string disp_str;
      ...
      $sformat(disp_str,"Type of env class is :
%s",env.env_c_type.name());
      `vmm_note(log,disp_str);
   end
endprogram
```

## `foreach_vmm_object()

Shorthand macro to iterate over all objects.

### SystemVerilog

```
`foreach_vmm_object(classtype, string name, vmm_root root);
```

### Description

This is a shorthand macro to iterate over all objects of a specified type and name, under a specified root.

### Example

```
class E extends vmm_object;
   ...
endclass
...
initial begin
   E e11 = new ("e11");
   vmm_object_iter my_iter;
   ...
   `foreach_vmm_object(vmm_object, "@%*", e11)
      begin
      ...
   end
end
```

## `foreach_vmm_object_in_namespace()

Shorthand macro to iterate over all objects of a specified type and name, within a specified namespace.

### SystemVerilog

```
`foreach_vmm_object_in_namespace(classtype, string name,
   string space, vmm_root root);
```

### Description

Shorthand macro to iterate over all objects of a specified type with the specified name, in the specified namespace under a specified root.

### Example

```
class C extends vmm_object;
   function new(string name, vmm_object parent=null);
      super.new(parent, name);
      ...
      vmm_object::create_namespace("NS1",
         vmm_object::IN_BY_DEFAULT);
      ...
   endfunction
endclass

C c1 = new("c1");
int I;

initial begin
   `foreach_vmm_object_in_namespace(vmm_object, "@%*",
         "NS1", c1)
      begin
      end
end
```

# vmm_object_iter

This is the `vmm_object` hierarchy traversal iterator class.

The `vmm_object_iter` class traverses the hierarchy rooted at the specified object, looking for objects whose relative hierarchical name matches the specified name. Beginning at a specific object, you can traverse through the hierarchy through the different methods like the `first()` and `next() methods`.

## Example

```
class E extends vmm_object;
   ...
endclass
...
initial begin
   E e11 = new ("e1");
   vmm_object obj;
   vmm_object_iter  iter = new (e11, "/a1/");
   ...
   obj = iter.first();
   while (obj != null)
      begin
         ...
         obj = iter.next;
      end
   ...
 end
```

## Summary


- vmm_object_iter::first() ..... page A-491
- vmm_object_iter::new() ..... page A-492
- vmm_object_iter::next() ..... page A-493

## vmm_object_iter::first()

Resets the state of the iterator to the first object.

### SystemVerilog

```
function vmm_object vmm_object_iter::first();
```

### Description

Resets the state of the iterator to the first object in the `vmm_object` hierarchy. Returns null, if the specified hierarchy contains no child objects.

### Example

```
class E extends vmm_object;
   ...
endclass
...
initial begin
   E e11 = new ("e1");
   vmm_object obj;
   vmm_object_iter  iter = new (e11, "/a1/");
   ...
   obj = iter.first();
   ...
end
```

## vmm_object_iter::new()

Instantiates an `vmm_object` iterator that traverses the hierarchy rooted at the specified root object.

### SystemVerilog

```
function new( vmm_object root = null, string  name = "",
    string space = "");
```

### Description

Traverses the hierarchy rooted at the specified root object, looking for objects whose relative hierarchical name in the specified namespace matches the specified name. The object name is relative to the specified root object. If no object is specified, traverses all hierarchies and the hierarchical name is absolute. The specified root (if any) is not included in the iteration.

### Example

```
/ Match pattern - /a1/, with root object e11 vmm_object_iter
   iter = new (e11, "/a1/" );
```

## vmm_object_iter::next()

### SystemVerilog

```
function vmm_object vmm_object_iter::next();
```

### Description

Returns the next object in the `vmm_object` hierarchy. Returns null, if there are no more child objects. Objects are traversed depth first.

### Example

```
class E extends vmm_object;
   ...
endclass
...
initial begin
   E e11 = new ("e1");
   vmm_object obj;
   vmm_object_iter  iter = new(e11, "/a1/");
   ...
   obj = iter.first();
   while (obj != null)
      begin
         ...
         obj = iter.next;
      end
   ...
end
```

# vmm_opts

Utility class that provides the facility to pass values from the command line during runtime, or from the source code, across hierarchies.

## Summary


- vmm_opts::get_bit() ..... page A-495
- vmm_opts::get_help() ..... page A-496
- vmm_opts::get_int() ..... page A-497
- vmm_opts::get_obj() ..... page A-498
- vmm_opts::get_object_bit() ..... page A-499
- vmm_opts::get_object_int() ..... page A-500
- vmm_opts::get_object_obj() ..... page A-502
- vmm_opts::get_object_range() ..... page A-504
- vmm_opts::get_object_string() ..... page A-506
- vmm_opts::get_range() ..... page A-508
- vmm_opts::get_string() ..... page A-510
- vmm_opts::set_bit() ..... page A-511
- vmm_opts::set_int() ..... page A-513
- vmm_opts::set_object() ..... page A-515
- vmm_opts::set_range() ..... page A-517
- vmm_opts::set_string() ..... page A-519
- `vmm_unit_config* ..... page A-521

## vmm_opts::get_bit()

Returns true, if specified option is set using the command-line. Otherwise, it returns false.

### SystemVerilog

```
static function bit vmm_opts::get_bit(string name,
   string doc = "", int verbosity = 0, string fname = "",
   int lineno = 0);
```

### Description

Returns true, if the argument name is specified on the command-line. Otherwise, it returns false. The option is specified using the command-line `+vmm_name` or `+vmm_opts+`*`name`*. You can specify a description of the option using `doc`, and the verbosity level of the option using `verbosity`. A verbosity value must be within the range 0 to 10. The `fname` and `lineno` arguments are used to track the file name and the line number, where the option is specified. These optional arguments are used for providing information to the user through `vmm_opts::get_help()`.

### Example

```
bit b;
b =  vmm_opts::get_bit(
   "FOO", "Value set for 'b' from command line");
```

Command line:

`simv +vmm_FOO` or `simv +vmm_opts+FOO`

## vmm_opts::get_help()

Displays the list of available or specified VMM runtime options.

### SystemVerilog

```
static function void vmm_opts::get_help(
                  vmm_object root = null,
                  int verbosity = 0);
```

### Description

Displays the known options used by the verification environment with the specified `vmm_object` hierarchy, with verbosity lower than or equal to the absolute value of the specified verbosity. If no `vmm_unit` root is specified, the options used by all object hierarchies are displayed. A verbosity value must be within the range -10 to 10. If the specified verbosity value is negative, the hierarchical name of each `vmm_unit` instance that uses an option is also displayed.

### Example

```
vmm_opts::get_help(this_object);
```

## vmm_opts::get_int()

Returns an integer value, if specified using the command-line. Otherwise, it returns the default value.

### SystemVerilog

```
static function int vmm_opts::get_int(string name,
   int dflt = 0, string doc = "", int verbosity = 0,
   string fname = "", int lineno = 0);
```

### Description

Returns an integer value, if the argument name and its integer value are specified on the command line. Otherwise, returns the default value specified in the `dflt` argument. The option is specified using the command line `+vmm_name=value` or `+vmm_opts+name=`*`value`*. You can specify a description of the option using `doc`, and the verbosity level of the option using `verbosity`. A verbosity value must be within the range 0 to 10. The `fname` and `lineno` arguments are used to track the file name and the line number, where the option is specified. These optional arguments are used to provide information through the `vmm_opts::get_help() method`.

### Example

```
int i;
i = vmm_opts::get_int ("FOO", 0,
   "Value set for 'i' from command line");
```

Command line:

`simv +vmm_FOO=100` or `simv +vmm_opts+FOO=100`

## vmm_opts::get_obj()

Returns the `vmm_object` instance, if specified through the `vmm_opts::set_object()` method.

### SystemVerilog

```
static function vmm_object vmm_opts::get_obj(
     output bit is_set,
     input string name,
     input vmm_object dflt = null,
     input string fname = "",
     input int lineno = 0);
```

### Description

If an explicit value is specified, returns the globally named object type option and sets the `is_set` argument to true. If no object matches the expression specified by `name`, returns the default object specified by argument `dflt`. Object type options can only be set using the `vmm_opts::set_object()` method. The `fname` and `lineno` arguments can be used to track the file name and the line number where the `get_obj` is invoked from.

### Example

```
class A extends vmm_object;
endclass

initial begin
   A a = new ("a");
   vmm_object obj;
   bit is_set;
   obj = vmm_opts :: get_obj(is_set, "OBJ", a);
end
```

## vmm_opts::get_object_bit()

Returns true, if the named option is set for the hierarchy. Otherwise, it returns false.

### SystemVerilog

```
static function bit vmm_opts::get_object_bit(output bit
   is_set, input vmm_object obj, string name, string doc =
   "", int verbosity = 0, string fname = "", int lineno = 0);
```

### Description

If an explicit value is specified, returns the named boolean type option for the specified object instance, and sets the `is_set` argument to true. You can specify a description of the option using `doc`, and the verbosity level of the option using `verbosity`. The verbosity value must be within the range 0 to 10, with 10 being the highest. The `fname` and `lineno` arguments are used to track the file name and the line number, where the option is specified. These optional arguments are used to provide information to the user through the `vmm_opts::get_help()` method.

### Example

```
class B extends vmm_object;
   bit foo, is_set;
   function new(string name, vmm_object parent=null);
   foo = vmm_opts::get_object_bit(is_set, this, "FOO",
      "SET foo value", 0);
   endfunction
endclass
```

Command line:

```
simv +vmm_FOO@A:%:b
```

## vmm_opts::get_object_int()

Returns an integer value, if the named integer option is set for the hierarchy. Otherwise, it returns the default value.

### SystemVerilog

```
static function int vmm_opts::get_object_int(
     output bit is_set,
     input vmm_object obj,
     input string name,
     input int dflt = 0,
     input string doc = "",
     input int verbosity = 0,
     input string fname = "",
     input int lineno = 0);
```

### Description

If an explicit value is specified, returns the named integer type option for the specified object instance and sets the `is_set` argument to true. You can specify a description of the option using `doc`, and the verbosity level of the option using `verbosity`. The verbosity value must be within the range 0 to 10. The `fname` and `lineno` arguments are used to track the file name and the line number, where the option is specified. These optional arguments are used to provide information through the `vmm_opts::get_help()` method.

### Example

```
class B extends vmm_object;
   int foo;
   function new(string name, vmm_object parent=null);
      bit is_set;
      super.new(parent,name);
      foo = vmm_opts::get_object_int(is_set, this, "FOO",
```

```
         2, "SET foo value", 0);
   endfunction
endclass
```

Command line:

```
simv +vmm_FOO=25@%:X:b
```

## vmm_opts::get_object_obj()

Returns the `vmm_object` instance for the specified hierarchical name.

### SystemVerilog

```
static function vmm_object get_object_obj(
     output bit is_set,
     input vmm_object obj,
     input string name,
     input vmm_object dflt = null,
     input  string    doc = "",
    input  int       verbosity = 0,
    input  string    fname  = "",
    input  int       lineno = 0);
```

### Description

Iif an explicit value is specified, returns the named object type option for the specified object instance and set the `is_set` argument to true. If no object matches the expression specified by `name`, returns the default object specified by argument `dflt`. You can specify a description of the option using `doc`, and the verbosity level of the option using `verbosity`. Object type options can only be set using the `vmm_opts::set_object()` method. The `fname` and `lineno` arguments can be used to track the file name and the line number where the `get_object_obj` is invoked from.

### Example

```
class A extends vmm_object;
   int foo = 11;
   function new( vmm_object parent=null, string name);
      super.new(parent, name);
   endfunction
```

```
endclass

class B extends vmm_object;
   A a1, a2;
   function new(vmm_object parent=null, string name);
      bit is_set;
      super.new(parent, name);
      a1 = new(null, "a1");
      a2 = new(null, "a2");
      a2.foo = 22;
      $cast(a1, vmm_opts::get_object_obj(is_set, this,
         "OBJ_F1",a2,"SET OBJ", 0));
   endfunction
endclass
```

## vmm_opts::get_object_range()

Returns the integer range for the specified hierarchy.

### SystemVerilog

```
static function void vmm_opts::get_object_range(
     output bit is_set,
     input vmm_object obj,
     input string name,
     output int min,max,
     input int dflt_min, dflt_max,
     input string doc = "", int verbosity = 0,
     input string fname = "",    int lineno = 0);
```

### Description

If an explicit range is specified, sets the *min* and *max* parameters to the values of the named integer-range-type option for the specified object instance, and sets the `is_set` argument to true. A range option is specified using the syntax `+vmm_name=[`*min*`:`*max*`]`. You can specify a description of the option using `doc`, and the verbosity level of the option using `verbosity`. A verbosity value must be within the range 0 to 10. The `fname` and `lineno` arguments are used to track the file name and the line number, where the option is specified. These optional arguments are used to provide information through the `vmm_opts::get_help() method`.

If no explicit values are provided for integer range of the specified hierarchy, sets the default range values to specified arguments `dflt_min` & `dflt_max` to the min and max arguments respectively. The fname and lineno arguments are used to track the file name and the line number where the get_object_range is invoked.

## Example

```
class B extends vmm_object;
   int min_val = -1;
   int max_val = -1;
   function new(string name, vmm_object parent=null);
      bit is_set;
      super.new(parent,name);
      vmm_opts::get_object_range(is_set, this,
      "FOO", min_val, max_val,-1,-1, "SET foo value", 0);
   endfunction
endclass
```

Command line:

```
simv +vmm_FOO=[5:10]@%:X:b
```

## vmm_opts::get_object_string()

Returns a string value, if the named string option is set for the hierarchy. Otherwise, it returns the default value.

### SystemVerilog

```
static function string get_object_string(output bit is_set,
input vmm_object obj, string name, string dflt, string doc
= "", int verbosity = 0, string fname = "", int lineno = 0);
```

### Description

If an explicit value is specified, returns the named string type option for the specified object instance, and sets the `is_set` argument to true. If no explicit value is specified, specified default string name `dftl` is assigned to string `name`. You can specify a description of the option using `doc`, and the verbosity level of the option using `verbosity`. The verbosity value must be within the range 0 to 10. The `fname` and `lineno` arguments are used to track the file name and the line number, where the option is specified. These optional arguments are used to provide information through the `vmm_opts::get_help()` method. The `fname` and `lineno` arguments are used to track the file name and the line number where the `get_object_string` is invoked.

### Example

```
class B extends vmm_object;
   string foo="ZERO";
   function new(string name, vmm_object parent=null);
      bit is_set;
      super.new(parent,name);
      foo = vmm_opts::get_object_string(is_set, this,
         "FOO", "DEF_VAL", "SET foo value", 0);
```

```
   endfunction
endclass
```

Command line:

```
simv +vmm_FOO=HELLO@%:X:b
```

## vmm_opts::get_range()

Returns an integer range, if specified using the command-line. Otherwise, it returns the default range.

### SystemVerilog

```
static function void vmm_opts::get_range(string name,
   output int min,max, input int dflt_min, dflt_max,
   string doc = "", int verbosity = 0, string fname = "",
   int lineno = 0);
```

### Description

Returns the named integer range option. A range option is specified using the syntax `+vmm_name=[`*`min`*`:`*`max`*`]` or `+vmm_opts+`*`name`*`=[`*`min`*`:`*`max`*`]`. You can specify a description of the option using `doc`, and the verbosity level of the option using `verbosity`. A verbosity value must be within the range 0 to 10. The `fname` and `lineno` arguments are used to track the file name and the line number, where the option is specified. These optional arguments are used to provide information through the `vmm_opts::get_help()` method.

If no explicit values are provided for integer range of the specified hierarchy, sets the default range values to specified arguments `dflt_min` & `dflt_max` to the min and max arguments respectively. The fname and lineno arguments are used to track the file name and the line number where the get_object_range is invoked.

### Example

```
int min_val;
int max_val;
```

```
vmm_opts::get_range("FOO", min_val, max_val,
   -1, -1, "SET range", 0);
```

Command line:

`simv +vmm_FOO=[5:10]` or `simv +vmm_opts+FOO=[5:10]`

## vmm_opts::get_string()

Returns the string value, if specified using the command-line. Otherwise, it returns the default value.

### SystemVerilog

```
static function string vmm_opts::get_string(string name,
   string dflt, string doc = "", int verbosity = 0,
   string fname = "", int lineno = 0);
```

### Description

Returns string value, if the argument name and its string value are specified on the command-line. Otherwise, it returns the default value specified in the `dflt` argument. The option is specified using the command line `+vmm_name=value` or `+vmm_opts+name=`*`value`*. You can specify a description of the option using `doc`, and the verbosity level of the option using `verbosity`. A verbosity value must be within the range 0 to 10. The `fname` and `lineno` arguments are used to track the file name and the line number, where the option is specified. These optional arguments are used to providing information through the `vmm_opts::get_help()` method.

### Example

```
string str;
str = vmm_opts :: get_string ("FOO", "DEF",
   "str value from command line");
```

Command line:

`simv +vmm_FOO=HELLO` or `simv +vmm_opts+FOO=HELLO`

## vmm_opts::set_bit()

Sets the hierarchically named boolean type option.

### SystemVerilog

```
static function void   vmm_opts::set_bit(string  name,
                                         bit val,
                                         vmm_unit root = null,
                                         string fname = "",
                                         int lineno = 0);

With +define NO_VMM12

static function void   vmm_opts::set_bit(string  name,
                                         bit val,
                                         string fname = "",
                                         int lineno = 0);
```

### Description

Sets the hierarchically named boolean type option for the specified `vmm_object` instances as specified by `val`. If no `vmm_unit` root is specified, the hierarchical option name is assumed to be absolute. The argument `name` can be a pattern. When `vmm_opts::get_object_bit()` is called in any object whose hierarchical name matches the pattern, the option is set for that boolean variable. The `fname` and `lineno` arguments are used to track the file name and the line number, where the option is specified from.

### Example

```
class B extends vmm_object;
   bit foo;
   function new(string name, vmm_object parent=null);
```

```
      bit is_set;
      super.new(parent,name);
      foo = vmm_opts::get_object_bit(is_set, this, "FOO",
         "SET foo value", 0);
   endfunction
endclass

B b2;
initial begin
   vmm_opts::set_bit("b2:FOO",null);
   b2 = new("b2", null);
end
```

## vmm_opts::set_int()

Sets the hierarchically named integer type option.

### SystemVerilog

```
static function void vmm_opts::set_int(string  name,
                                       int val,
                                       vmm_unit root = null,
                                       string fname = "",
                                       int lineno = 0);

With +define NO_VMM12
static function void vmm_opts::set_int(string  name,
                                       int val,
                                       string fname = "",
                                       int lineno = 0);
```

### Description

Sets the hierarchically named integer type option for the specified `vmm_object` instances as specified by `val`. If no `vmm_unit` root is specified, the hierarchical option name is assumed to be absolute. The argument `name` can be a pattern. When `vmm_opts::get_object_bit()` is called in any object whose hierarchical name matches the pattern, the option is set for that integer variable. The `fname` and `lineno` arguments are used to track the file name and the line number, where the option is specified from.

### Example

```
class A extends vmm_object;
   int a_foo;
   function new(vmm_object parent=null, string name);
      bit is_set;
```

```
      super.new(parent, name);
      a_foo = vmm_opts::get_object_int(is_set, this,
         "A_FOO", 2 , "SET a_foo value", 0);
   endfunction
endclass

class D extends vmm_object;
   A a1;
   ...
endclass

initial  begin
   D d2;
   vmm_opts::set_int("d2:a1:A_FOO", 99,null);
   d2 = new (null, "d2");
end
```

## vmm_opts::set_object()

Sets the hierarchically named `vmm_object` type option.

### SystemVerilog

```
static function void   vmm_opts::set_object(string  name,
                                            vmm_object obj,
                                              vmm_unit root = null,
                                            string fname = "",
                                            int lineno = 0);

With +define NO_VMM12

static function void   vmm_opts::set_object(string  name,
                                              vmm_object obj,
                                              string fname = "",
                                              int lineno = 0);
```

### Description

Sets the hierarchically named type-specific option for the specified `vmm_object` instances. If no `vmm_unit` root is specified, the hierarchical option name is assumed to be absolute. When called from the `vmm_unit::configure_ph()` method, the root unit must always be specified as `this`, because `vmm_unit` instances can only configure lower-level instances during the `configure` phase. The hierarchical option name is specified by prefixing the option name with a hierarchical `vmm_unit` name and a colon (:).

The hierarchical option name may be specified using a match pattern or a regular expression, except for the last part of the hierarchical name (the name of the option itself). The hierarchical option name may specify a namespace. An error is reported, if the option value is not eventually used.

The `fname` and `lineno` arguments are used to track the file name and the line number, where the option is specified from.

## Example

```
class A extends vmm_object;
   int foo = 11;
   function new( vmm_object parent=null, string name);
      bit is_set;
      super.new(parent, name);
   endfunction
endclass

class B extends vmm_object;
   A a1;
   A a2;
   function new(vmm_object parent=null, string name);
      bit is_set;
      super.new(parent, name);
      a1 = new(null, "a1");
      a2 = new(null, "a2");
      a2.foo = 22;
      $cast(a1, vmm_opts::get_object_obj(is_set, this,
         "OBJ_F1",a2,"SET OBJ", 0));
   endfunction
endclass

B b2;
A a3;
initial begin
   a3 = new(null, "a3");
   a3.foo = 99;
   vmm_opts::set_object("b2:OBJ_F1", a3,null,,);
   b2 = new(null, "b2");
end
```

## vmm_opts::set_range()

Sets the hierarchically named integer range type option.

### SystemVerilog

```
static function void vmm_opts::set_range(string  name,
                                         int min, max,
                                          vmm_unit root = null,
                                         string fname = "",
                                         int lineno = 0);

With +define NO_VMM12

static function void vmm_opts::set_range(string  name,
                                          int min, max,
                                          string fname = "",
                                          int lineno = 0);
```

### Description

Sets the hierarchically named integer range type option, for the specified `vmm_object` instances. If no `vmm_unit` root is specified, then the hierarchical option name is assumed to be absolute. The *`name`* argument can be a pattern. When `vmm_opts::get_object_range()` is called in an object whose hierarchical name matches the pattern, then *`min`* and *`max`* are returned.

The `fname` and `lineno` arguments are used to track the file name and the line number, where the range is specified from.

### Example

```
class B extends vmm_object;
```

```
   int min_val = -1;
   int max_val = -1;
   function new(string name, vmm_object parent=null);
      bit is_set;
      super.new(parent,name);
      vmm_opts::get_object_range(is_set, this,
         "FOO",min_val, max_val, -1,-1, "SET foo value", 0);
   endfunction
endclass
initial begin
   B b2;
   vmm_opts::set_range("b2:FOO", 1, 99, null);
   b2 = new("b2", null);
end
```

## vmm_opts::set_string()

Sets the hierarchical named string type option.

### SystemVerilog

```
static function void vmm_opts::set_string(string  name,
                                          string val,
                                            vmm_unit root = null,
                                          string fname = "",
                                          int lineno = 0);

With +define NO_VMM12

static function void vmm_opts::set_string(string  name,
                                            string val,
                                            string fname = "",
                                            int lineno = 0);
```

### Description

Sets the hierarchically named string type option, for the specified `vmm_object` instances as specified by `name`. If no `vmm_unit` root is specified, then the hierarchical option name is assumed to be absolute. The argument *`name`* can be a pattern. When the `vmm_opts::get_object_string()` method is called in any object whose hierarchical name matches the pattern, then *`val`* is returned for that string variable.

The `fname` and `lineno` arguments are used to track the file name and the line number, where the option is specified from.

### Example

```
class B extends vmm_object;
```

```
   string foo="ZERO";
   function new(string name, vmm_object parent=null);
      bit is_set;
      super.new(parent,name);
      foo = vmm_opts::get_object_string(is_set, this,
         "FOO","DEF_VAL", "SET foo value", 0);
   endfunction
endclass

initial begin
   B b2;
   vmm_opts::set_string("b2:FOO", "NEW_VAL", null);
   b2 = new("b2", null);
end
```

## ‘vmm_unit_config*

This section describes the following macros:

- “`vmm_unit_config_begin( <classname> )”
- “`vmm_unit_config_boolean(name, descr, verbosity, attribute)”
- “`vmm_unit_config_end( <classname> )”
- “`vmm_unit_config_int(name, dflt, descr, verbosity, attribute)”
- “`vmm_unit_config_obj(name, dflt, descr, verbosity, attribute)”
- “`vmm_unit_config_rand_boolean(name, descr, verbosity, attribute)”
- “`vmm_unit_config_rand_int(name, dflt, descr, verbosity, attribute)”
- “`vmm_unit_config_rand_obj(name, dflt, descr, verbosity, attribute)”
- “`vmm_unit_config_string(name, dflt, descr, verbosity, attribute)”

### `vmm_unit_config_begin( <classname> )

Macro, which indicates the beginning of the structural configuration parameters setting in the `vmm_unit::configure_ph()` phase.

### `vmm_unit_config_boolean(name, descr, verbosity, attribute)

Macro for setting Boolean value to the variable, with the name specified in the argument.

It internally calls `vmm_opts::get_object_bit,` which uses description and verbosity arguments as well.

The `attribute` argument is for future enhancements.

**`vmm_unit_config_end( <classname> )**

Macro, which indicates the end of the structural configuration.

**`vmm_unit_config_int(name, dflt, descr, verbosity, attribute)**

Macro for setting integer value to the variable, with the name specified in the argument.

It internally calls `vmm_opts::get_object_int,` which uses default value, description, and verbosity arguments as well.

The `attribute` argument is for future enhancements.

**`vmm_unit_config_obj(name, dflt, descr, verbosity, attribute)**

Macro for setting object value to the variable, with the name specified in the argument.

It internally calls `vmm_opts::get_object_obj,` which uses default value and verbosity arguments as well.

The `description` and `attribute` arguments are for future enhancements.

**`vmm_unit_config_rand_boolean(name, descr, verbosity, attribute)**

Macro for setting Boolean value to the variable, with the name specified in the argument.

It internally calls `vmm_opts::get_object_bit,` which uses description and verbosity arguments as well.

It also sets the `rand_mode` of the variable to 0, so that the value set through configuration will not change due to randomization.

The `attribute` argument is for future enhancements.

**`vmm_unit_config_rand_int(name, dflt, descr, verbosity, attribute)**

Macro for setting integer value to the variable, with the name specified in the argument.

It internally calls `vmm_opts::get_object_int,` which uses default value, description, and verbosity arguments as well.

It also sets the `rand_mode` of the variable to 0, so that the value set through configuration will not change due to randomization.

The `attribute` argument is for future enhancements.

**`vmm_unit_config_rand_obj(name, dflt, descr, verbosity, attribute)**

Macro for setting object value to the variable, with the name specified in the argument.

It internally calls `vmm_opts::get_object_obj,` which uses default value and verbosity arguments as well.

It also sets the `rand_mode` of the variable to 0, so that the value set through configuration will not change due to randomization.

The `description` and `attribute` arguments are for future enhancements.

### `vmm_unit_config_string(name, dflt, descr, verbosity, attribute)

Macro for setting string value to the variable, with the name specified in the argument.

It internally calls `vmm_opts::get_object_string,` which uses default value, description, and verbosity arguments as well.

The `attribute` argument is for future enhancements.

*Example A-171*

```
class my_driver extends vmm_xactor;
   string      my_name;
   rand int    my_int;
   bit         my_bool;

   `vmm_unit_config_begin(my_driver)
   `vmm_unit_config_string(my_name, "HELLO",
      "Sets string value", 0, DO_ALL)
   `vmm_unit_config_rand_int(my_int, 5, "Sets int value and
      switches off rand_mode", 0, DO_ALL)
   `vmm_unit_config_boolean(my_bool, "Sets/Resets boolean
      value", 0, DO_ALL)
   `vmm_unit_config_end(my_driver)

endclass
```

# B

# Standard Library Classes (Part 2)

This appendix provides detailed information about the OpenVera and SystemVerilog classes that compose the VMM Standard Library. The functionality of OpenVera and SystemVerilog classes is identical, except for the following difference:

- OpenVera methods have a prefix of `rvm`
- SystemVerilog methods have a prefix of `vmm`

Note:

Each method, explained in this appendix, uses the SystemVerilog name in the heading to introduce it. Additionally, there are a few instances where a `_t` suffix is appended to indicate that it may be a blocking method.

Usage examples are specified in a single language, but that should not prevent the use of the other language, as both the languages are almost identical. Rather than providing usage examples that are almost identical, this appendix provides different examples for each language.

The classes are documented in alphabetical order. The methods in each class are documented in a logical order, where methods that accomplish similar results are documented sequentially. A summary of all available methods, with cross-references to the page where their detailed documentation can be found, is provided at the beginning of each class specification.

## VMM Standard Library Class List


- “vmm_phase”
- “vmm_phase_def”
- “vmm_rtl_config_DW_format”
- “vmm_rtl_config”
- “vmm_rtl_config_file_format”
- “vmm_scenario”
- “vmm_scenario_gen#(T, text)”
- “<class-name>_scenario”
- “<class-name>_atomic_scenario”
- “<class-name>_scenario_election”
- “<class-name>_scenario_gen_callbacks”

- “vmm_scheduler”
- “vmm_scheduler_election”
- “vmm_ss_scenario#(T)”
- “vmm_simulation”
- “vmm_subenv”
- “vmm_test”
- “vmm_test_registry”
- “vmm_timeline”
- “vmm_timeline_callbacks”
- “vmm_tlm”
- “vmm_tlm_generic_payload”
- “vmm_tlm_analysis_port#(I,D)”
- “vmm_tlm_analysis_export#(T,D)”
- “‘vmm_tlm_analysis_export(SUFFIX)”
- “vmm_tlm_b_transport_export#(T,D)”
- “vmm_tlm_b_transport_port #(I,D)”
- “vmm_tlm_export_base #(D,P)”
- “vmm_tlm_nb_transport_bw_export#(T,D,P)”
- “vmm_tlm_nb_transport_bw_port#(I,D,P)”
- “vmm_tlm_nb_transport_export#(T,D,P)”
- “vmm_tlm_nb_transport_fw_export#(T,D,P)”

- “vmm_tlm_nb_transport_fw_port#(I,D,P)”
- “vmm_tlm_nb_transport_port#(I,D,P)”
- “vmm_tlm_port_base#(D,P)”
- “vmm_tlm_initiator_socket#(I,D,P)”
- “vmm_tlm_target_socket#(T,D,P)”
- “vmm_unit”
- “vmm_version”
- “vmm_voter”
- “vmm_xactor”
- “vmm_xactor_callbacks”
- “vmm_xactor_iter”

# vmm_phase

The `vmm_phase` class is used as a container for phase descriptors, and their associated statistical information.

## Summary


- vmm_phase::completed ..... page B-6
- vmm_phase::started ..... page B-7
- vmm_phase::get_name() ..... page B-8
- vmm_phase::get_timeline() ..... page B-9
- vmm_phase::is_aborted() ..... page B-10
- vmm_phase::is_done() ..... page B-12
- vmm_phase::is_running() ..... page B-13
- vmm_phase::is_skipped() ..... page B-14
- vmm_phase::next_phase() ..... page B-16
- vmm_phase::previous_phase() ..... page B-17

## vmm_phase::completed

Phase execution completion event.

### Description

This event is triggered when the execution of this phase is completed.

### Example

```
vmm_timeline top;
vmm_phase ph;

initial begin
   top = new("top", "top");
   ph = top.get_phase("connect");
   @(ph.completed);
   `vmm_log (log, "Completed execution of phase connect");
   ...
end
```

## vmm_phase::started

Phase execution start event.

### Description

This event is triggered when the execution of this phase starts.

### Example

```
vmm_timeline top;
vmm_phase ph;

initial begin
   top = new("top", "top");
   ph = top.get_phase("connect");
   ...
   @(ph.started);
   `vmm_note(log," connect phase execution started");
   ...
end
```

## vmm_phase::get_name()

Method to get the phase descriptor name.

### SystemVerilog

```
function string vmm_phase::get_name()
```

### Description

Returns the name of the phase descriptor.

### Example

```
vmm_timeline top;
vmm_phase ph;
string ph_name;

initial begin
   top = new("top", "top");
   ph = top.get_phase("connect");
   ...
   ph_name = ph.get_name();  //returns string "connect"
   ...
end
```

## vmm_phase::get_timeline()

Method to get the enclosing timeline.

### SystemVerilog

```
function vmm_timeline vmm_phase::get_timeline()
```

### Description

Returns the timeline, which contains this phase.

### Example

```
vmm_timeline top;
vmm_phase ph;

initial begin
   vmm_timeline t;
   top = new("top", "top");
   ph = top.get_phase("connect");
   ...
   t = ph.get_timeline;
   ...
end
```

## vmm_phase::is_aborted()

Method to check aborted status of the phase.

### SystemVerilog

```
function int vmm_phase::is_aborted()
```

### Description

Returns the number of times that the phase is aborted.

### Example

```
class myTest  extends vmm_timeline;
   function new(string name, string inst,
         vmm_object parent = null);
      super.new(name, inst, parent);
   endfunction

   task reset_ph;
      $display("%t:Starting Reset", $time);
      #5;
      $display("%t:Finishing Reset", $time);
   endtask

   task training_ph;
      #5;
   endtask

   task run_ph;
      #5;
   endtask

endclass

vmm_log log = new("test", "main");
myTest top;
```

```
initial begin
   vmm_phase ph_reset;

   top = new("top", "top");
   ph_reset = top.get_phase("reset");

   fork
      top.run_phase();
   join_none
   #7 top.abort_phase("training"); //aborting training
   #1 top.reset_to_phase("reset"); //aborting run
   #1 top.jump_to_phase("run"); //aborting reset,
                        // skipping training-start_of_test

   #10;

   if(ph_reset.is_aborted() != 2)
      `vmm_error(log,`vmm_sformatf(
        $psprintf("Expected reset to abort 2 times,
           is_aborted returns %d",ph_reset.is_aborted))
        );
```

## vmm_phase::is_done()

Method to check completion status of the phase.

### SystemVerilog

```
function int vmm_phase::is_done()
```

### Description

Returns the number of times that the phase is completed.

## vmm_phase::is_running()

Method to get execution status of the phase.

### SystemVerilog

```
function bit vmm_phase::is_running()
```

### Description

Returns true, if the phase is currently being executed. Always returns false for function phases, unless called from within the phase implementation function itself.

### Example

```
vmm_timeline top;
vmm_phase ph;

initial
   begin
      top = new("top", "top");
      ph = top.get_phase("connect");
      ...
      wait(ph.is_running == 0);
      ...
   end
```

## vmm_phase::is_skipped()

Returns the number of times that the phase is skipped.

### SystemVerilog

```
function int vmm_phase::is_skipped()
```

### Description

Returns the number of times that the phase is skipped.

### Example

```
class myTest  extends vmm_timeline;
   function new(string name, string inst,
         vmm_object parent = null);
      super.new(name, inst, parent);
   endfunction

   task reset_ph;
      $display("%t:Starting Reset", $time);
      #5;
      $display("%t:Finishing Reset", $time);
   endtask

   task training_ph;
      #5;
   endtask

   task run_ph;
      #5;
   endtask

endclass

vmm_log log = new("test", "main");
myTest top;
```

```
initial begin
   vmm_phase ph_training;
   top = new("top", "top");
   ph_training = top.get_phase("training");

   fork
      top.run_phase();
   join_none
   #9 top.jump_to_phase("run");  //aborting reset,
                        //skipping training-start_of_test
   #10;

   if(ph_training.is_skipped() != 1)
      `vmm_error(log,`vmm_sformatf(
         $psprintf("Expected training to abort 1 times,
         is_skipped returns %d",ph_training.is_skipped))
      );
```

## vmm_phase::next_phase()

Method to get the following phase descriptor.

### SystemVerilog

```
function vmm_phase vmm_phase::next_phase()
```

### Description

Returns the following phase in the timeline containing this phase. Returns null, if this is the last phase in the timeline.

### Example

```
vmm_timeline top;
vmm_phase ph;

initial begin
   vmm_phase nx_ph;
   top = new("top", "top");
   ph = top.get_phase("connect");
   ...
   nx_ph = ph.next_phase(); //returns phase configure_test
   `vmm_note(log,`vmm_sformatf("
      %s will execute after connect",nx_ph.get_name());
   ...
end
```

## vmm_phase::previous_phase()

Method to get the preceding phase descriptor.

### SystemVerilog

```
function vmm_phase vmm_phase::previous_phase()
```

### Description

Returns the preceding phase in the timeline containing this phase. Returns null, if this is the first phase in the timeline.

### Example

```
vmm_timeline top;
vmm_phase ph;

initial begin
   vmm_phase prv_ph;
   top = new("top", "top");
   ph = top.get_phase("connect");
   ...
   prv_ph = ph.previous_phase(); //returns phase configure
   `vmm_note(log,`vmm_sformatf(
      "connect will execute after %s ",prv_ph.get_name());
   ...
end
```

# vmm_phase_def

The `vmm_phase_def` virtual class is extended to create a user-defined phase.

## Summary


- vmm_bottomup_function_phase_def ..... page B-19
- vmm_bottomup_function_phase_def::do_function_phase() page B-20
- vmm_fork_task_phase_def#(T) ..... page B-21
- vmm_fork_task_phase_def::do_task_phase() ..... page B-22
- vmm_null_phase_def ..... page B-23
- vmm_phase_def::is_function_phase() ..... page B-24
- vmm_phase_def::is_task_phase() ..... page B-25
- vmm_phase_def::run_function_phase() ..... page B-26
- vmm_phase_def::run_task_phase() ..... page B-27
- vmm_reset_xactor_phase_def ..... page B-28
- vmm_start_xactor_phase_def ..... page B-30
- vmm_stop_xactor_phase_def ..... page B-32
- vmm_topdown_function_phase_def ..... page B-34
- vmm_topdown_function_phase_def::do_function_phase() page B-35
- vmm_xactor_phase_def ..... page B-36

## vmm_bottomup_function_phase_def

Predefined bottom-up phase definition.

### SystemVerilog

```
class vmm_bottomup_function_phase_def #(type T)
   extends vmm_function_phase_def
```

### Description

Implements the `vmm_phase_def::run_function_phase()`. To call the `vmm_bottomup_function_phase_def::do_function_phase()` method on any object of specified type, within the `vmm_object` hierarchy under the specified root, in a bottom-up order.

## vmm_bottomup_function_phase_def::do_function_phase()

Method to execute an object for particular phase execution.

### SystemVerilog

```
virtual function void
 vmm_bottomup_function_phase_def::do_function_phase(T obj)
```

### Description

Implementation of the function phase on an object of the specified type. You can choose to execute some non-delay processes of a specified object in this method, of a new phase definition class extended from this class.

### Example

```
class udf_phase_def extends
      vmm_bottomup_function_phase_def;
   function void do_function_phase(vmm_unit un1);
      un1.my_method();
   endfunction
endclass
```

## vmm_fork_task_phase_def#(T)

Predefined  task based phase definition.

```
class vmm_fork_task_phase_def #(type T) extends
   vmm_task_phase_def
```

### SystemVerilog

### Description

Implements the `vmm_phase_def::run_task_phase()`. To make a call to the `vmm_fork_task_phase_def::do_task_phase()` method on any object of a specified type, within the `vmm_object` hierarchy, under the specified root in a top-down order.

## vmm_fork_task_phase_def::do_task_phase()

Method to execute on object for particular phase execution.

### SystemVerilog

```
virtual task vmm_fork_task_phase_def::do_task_phase(T obj)
```

### Description

Implementation of the task phase on an object of the specified type. You can choose to execute time-consuming processes in this method, of a new phase definition class extended from this class.

### Example

```
class udf_phase_def extends vmm_fork_task _phase_def;
   task do_task_phase(vmm_unit un1);
     un1.my_method();
   endtask
endclass
```

## vmm_null_phase_def

Predefined null phase definition.

### SystemVerilog

```
class vmm_null_phase_def extends vmm_phase_def
```

### Description

Implements empty `vmm_phase_def::run_function_phase()` and `vmm_phase_def::run_task_phase()`. Typically used to override a predefined phase to skip its predefined implementation for a specific `vmm_unit` instance.

### Example

```
class myphase_def extends
   vmm_null_phase_def #(groupExtension);
endclass : myphase_def
myphase_def null_ph = new();
group_extension m1 = new("groupExtension","m1");
`void(m1.override_phase("configure",null_ph ));
//nothing to de done for this component in configure phase
```

## vmm_phase_def::is_function_phase()

Method to check the type of phase definition (check if it is a function).

### SystemVerilog

```
virtual function bit vmm_phase_def::is_function_phase()
```

### Description

Returns true, if this phase is executed by calling the `vmm_phase_def::run_function_phase()` method. Otherwise, it returns false.

### Example

```
virtual class user_function_phase_def #(
      user_function_phase_def) extends
      vmm_topdown_function_phase_def;
   function bit is_function_phase();
      return 1;
   endfunction:is_function_phase
endclass
```

## vmm_phase_def::is_task_phase()

Method to check type of phase definition (check if it is a task).

### SystemVerilog

```
virtual function bit vmm_phase_def::is_task_phase()
```

### Description

Returns true, if this phase is executed by calling the `vmm_phase_def::run_task_phase()` method. Otherwise, it returns false.

### Example

```
virtual class user_task_phase_def #( user_task_phase_def)
      extends vmm_fork_task_phase_def;
   function bit is_task_phase();
      return 1;
   endfunction:is_task_phase
endclass
```

## vmm_phase_def::run_function_phase()

Method to execute phase definition, used by timeline.

### SystemVerilog

```
virtual function void run_function_phase(string     name,
                                         vmm_object obj,
                                        vmm_log log);
```

### Description

Executes the function phase, under the specified name on the specified object. This method must be overridden, if the `vmm_phase_def::is_function_phase()` method returns true.

The argument `log` is the message interface instance to be used by the phase for reporting information.

### Example

```
virtual class user_function_phase_def #(
      user_function_phase_def)
      extends vmm_topdown_function_phase_def;
   function bit is_function_phase();
      return 1;
   endfunction:is_function_phase

   function run_function_phase(string name,
         vmm_object root, vmm_log log);
      `vmm_note(log,`vmm_sformatf(
         "Executing phase %s for %s", name,
         root.get_object_name());
   endfuction
endclass
```

## vmm_phase_def::run_task_phase()

Method to execute phase definition, used by timeline.

### SystemVerilog

```
virtual task run_task_phase(string name,
                            vmm_object obj,
                            vmm_log log);
```

### Description

Executes the task phase, under the specified name on the specified root object. This method must be overridden if the `vmm_phase_def::is_task_phase()` method returns true.

The argument `log` is the message interface instance to be used by the phase for reporting information.

### Example

```
virtual class user_task_phase_def #( user_task_phase_def)
      extends vmm_fork_task_phase_def;
   function bit is_task_phase();
      return 1;
   endfunction:is_task_phase
   task run_task_phase(string name, vmm_object root,
         vmm_log log);
      `vmm_note(log,`vmm_sformatf(
         "Executing phase %s for %s", name,
         root.get_object_name());
   endtask

endclass
```

## vmm_reset_xactor_phase_def

Predefined `vmm_reset_xactor` phase definition class.

### SystemVerilog

```
class vmm_reset_xactor_phase_def extends
   vmm_xactor_phase_def;
```

### Description

Implements the `vmm_reset_xactor_phase_def::do_function_phase()`. This function calls the `reset_xactor()` function, on a specified object of type `vmm_xactor`.

### Example

```
class consumer extends vmm_xactor ;
   packet_channel in_chan;
   function new(string inst, packet_channel in_chan);
      super.new("consumer", inst);
      this.in_chan = in_chan;
   endfunction
   ...
   ...

class consumer_timeline #(string phase = "reset") extends
      vmm_timeline;
   `vmm_typename(consumer_timeline)
   consumer xactor;
   packet_channel   chan;

   function new (string inst, packet_channel  chan,
         vmm_unit parent = null);
      super.new(get_typename(),inst, parent);
      this.chan = chan;
```

```
   endfunction

   function void build_ph;
      xactor = new("xactor", chan);
      xactor.set_parent_object(this);
   endfunction

   function void connect_ph;
      vmm_reset_xactor_phase_def reset = new(
         "consumer","xactor");
      void'(this.insert_phase(phase,phase, reset));
   endfunction
   ...
   ...
endclass

consumer_timeline  #("reset") ctl = new("ctl", chan);
```

## vmm_start_xactor_phase_def

Predefined `vmm_start_xactor` phase definition class.

### SystemVerilog

```
class vmm_start_xactor_phase_def extends
   vmm_xactor_phase_def;
```

### Description

Implements the `vmm_start_xactor_phase_def::do_function_phase()`. This function calls the `start_xactor()` function, on specified object of type `vmm_xactor`.

### Example

```
class consumer extends vmm_xactor ;
   packet_channel   in_chan;
   function new(string inst, packet_channel in_chan);
      super.new("consumer", inst);
      this.in_chan = in_chan;
   endfunction
   ...
   ...

class consumer_timeline #(string phase = "start") extends
      vmm_timeline;
   `vmm_typename(consumer_timeline)
   consumer xactor;
   packet_channel chan;

   function new (string inst, packet_channel  chan,
         vmm_unit parent = null);
      super.new(get_typename(),inst, parent);
      this.chan = chan;
```

```
   endfunction

   function void build_ph;
      xactor = new("xactor", chan);
      xactor.set_parent_object(this);
   endfunction

   function void connect_ph;
      vmm_start_xactor_phase_def start = new(
         "consumer","xactor");
      void'(this.insert_phase(phase, phase, start));
   enfunction
   ...
   ...
   endclass

consumer_timeline  #("start") ctl = new("ctl", chan);
```

## vmm_stop_xactor_phase_def

Predefined `vmm_stop_xactor` phase definition class.

### SystemVerilog

```
class vmm_stop_xactor_phase_def extends
   vmm_xactor_phase_def;
```

### Description

Implements the `vmm_stop_xactor_phase_def::do_function_phase()`. This function calls the `stop_xactor()` function on a specified object of type `vmm_xactor`.

### Example

```
class consumer extends vmm_xactor ;
   packet_channel   in_chan;
   function new(string inst, packet_channel in_chan);
      super.new("consumer", inst);
      this.in_chan = in_chan;
   endfunction
   ...
   ...

class consumer_timeline #(string phase = "stop") extends
      vmm_timeline;
   `vmm_typename(consumer_timeline)
   consumer xactor;
   packet_channel chan;

   function new (string inst, packet_channel  chan,
         vmm_unit parent = null);
      super.new(get_typename(),inst, parent);
      this.chan = chan;
```

```
   endfunction

   function void build_ph;
      xactor = new("xactor", chan);
      xactor.set_parent_object(this);
   endfunction

   function void connect_ph;
      vmm_stop_xactor_phase_def stop = new(
         "consumer","xactor");
      void'(this.insert_phase(phase,phase, stop));
   endfunction
   ...
   ...
endclass

consumer_timeline  #("shutdown") ctl = new("ctl", chan);
```

## vmm_topdown_function_phase_def

Predefined top-down phase definition.

### SystemVerilog

```
class vmm_topdown_function_phase_def #(type T=vmm_object)
extends vmm_phase_def;
```

### Description

Implements the `vmm_phase_def::run_function_phase()`. To call the `vmm_topdown_function_phase_def::do_function_phase()` method on any object of specified type within the `vmm_object` hierarchy under the specified root in a top-down order.

## vmm_topdown_function_phase_def::do_function_phase()

Method to execute an object for particular phase execution.

### SystemVerilog

```
virtual function void
  vmm_topdown_function_phase_def::do_function_phase(T obj)
```

### Description

Implementation of the function phase on an object of the specified type.

You can choose to execute some non-delay processes of the specified object in this method, of a new phase definition class extended from this class.

### Example

```
class udf_phase_def extends vmm_topdown_function_phase_def;
   function void do_function_phase(vmm_unit un1);
      un1.my_method();
   endfunction
endclass
```

## vmm_xactor_phase_def

Predefined `vmm_xactor` phase definition class.

### SystemVerilog

```
class vmm_xactor_phase_def #(type T=vmm_xactor) extends
vmm_phase_def;
```

### Description

Implements the `vmm_xactor_phase_def::run_function_phase()`, to call the `vmm_xactor_phase_def::do_function_phase()` method on any object of specified type within the `vmm_object` hierarchy, with specified name or instance.

# vmm_rtl_config_DW_format

Predefined implementation for an RTL configuration parameter, using the DesignWare Implementation IP file format.

## SystemVerilog

```
class vmm_rtl_config_DW_format extends
   vmm_rtl_config_file_format
```

# vmm_rtl_config

This is the base class for RTL configuration and extends `vmm_object`. This class is for specifying RTL configuration parameters. A different class from other parameters that use the `vmm_opts` class is used, because these parameters must be defined at compile time and may not be modified at runtime.

## Example

```
class ahb_master_config extends vmm_rtl_config;
   rand  int addr_width;
   rand  bit mst_enable;
   string    kind = "MSTR";

   constraint cst_mst {
      addr_width == 64;
      mst_enable == 1;
   }
   `vmm_rtl_config_begin(ahb_master_config)
      `vmm_rtl_config_int(addr_width, mst_width)
      `vmm_rtl_config_boolean(mst_enable, mst_enable)
      `vmm_rtl_config_string(kind, kind)
   `vmm_rtl_config_end(ahb_master_config)

   function new(string name = "", vmm_rtl_config parent =
         null);
      super.new(name, parent);
   endfunction
endclass
```

## Summary


- vmm_rtl_config::build_config_ph() ..... page B-40
- vmm_rtl_config::default_file_fmt ..... page B-41
- vmm_rtl_config::file_fmt ..... page B-42
- vmm_rtl_config::get_config() ..... page B-43
- vmm_rtl_config::get_config_ph() ..... page B-44
- `vmm_rtl_config_* ..... page B-45
- vmm_rtl_config::map_to_name() ..... page B-46

- vmm_rtl_config::save_config_ph() ..... page B-48

## vmm_rtl_config::build_config_ph()

Builds RTL configuration parameters.

### SystemVerilog

```
virtual function void vmm_rtl_config::build_config_ph()
```

### Description

Builds the structure of RTL configuration parameters for hierarchical RTL designs.

### Example

```
class env_config extends vmm_rtl_config;
   rand ahb_master_config mst_cfg;
   rand ahb_slave_config  slv_cfg;
   ...
   function void build_config_ph();
      mst_cfg = new("mst_cfg", this);
      slv_cfg = new("slv_cfg", this);
   endfunction
   ...
endclass
```

## vmm_rtl_config::default_file_fmt

Default RTL configuration file format.

### SystemVerilog

```
static vmm_rtl_config_file_format
   vmm_rtl_config::default_file_fmt
```

### Description

Default RTL configuration file format writer or parser. Used if the `vmm_rtl_config::file_fmt` is null.

### Example

```
class def_rtl_config_file_format extends
   vmm_rtl_config_file_format;
endclass

intial begin
   def_rtl_config_file_format dflt_fmt = new();
   vmm_rtl_config::default_file_fmt = dflt_fmt;
end
```

## vmm_rtl_config::file_fmt

RTL configuration file format.

### SystemVerilog

```
protected vmm_rtl_config_file_format
   vmm_rtl_config::file_fmt
```

### Description

The RTL configuration file format writer or parser for this instance.

### Example

```
//protected vmm_rtl_config_file_format vmm_rtl_config ::
      file_fmt
class ahb_rtl_config_file_format extends
   vmm_rtl_config_file_format;
endclass

class env_config extends vmm_rtl_config;
   rand ahb_master_config mst_cfg;
   ahb_rtl_config_file_format ahb_file_fmt;

   function void build_config_ph();
      mst_cfg = new("mst_cfg", this);
      ahb_file_fmt = new;
      mst_cfg.file_fmt = ahb_file_format;
   endfunction
endclass
```

## vmm_rtl_config::get_config()

Returns a `vmm_rtl_config` object for the specified `vmm_object`.

### SystemVerilog

```
static function vmm_rtl_config::get_config(vmm_object obj,
   string fname = "", int lineno = 0)
```

### Description

Gets the instance of the specified class extended from the `vmm_rtl_config` class, whose hierarchical name in the “VMM RTL Config” namespace is identical to the hierarchical name of the specified object. This allows a component to retrieve its instance-configuration, without having to know where it is located in the testbench hierarchy.

The `fname` and `lineno` arguments are used to track the file name and the line number where `get_config` is invoked from.

Example

```
class ahb_master extends vmm_group;
   ahb_master_config cfg;
   function void configure_ph();
      $cast(cfg, vmm_rtl_config::get_config(this,
         `__FILE__, `__LINE__));
   endfunction
endclass
```

## vmm_rtl_config::get_config_ph()

Sets the RTL configuration parameters.

### SystemVerilog

```
virtual function void vmm_rtl_config::get_config_ph()
```

### Description

Reas a configuration file and sets the current value of members to the corresponding RTL configuration parameters. The filename may be computed using the value of the `+vmm_rtl_config` option, using the `vmm_opts::get_string("rtl_config”")` method and the hierarchical name of this `vmm_object` instance.

A default implementation of this method is created, if the `` `vmm_rtl_config_*() `` shorthand macros are used.

## ‘vmm_rtl_config_*

**`vmm_rtl_config_begin(classname)**
**`vmm_rtl_config_boolean(name, fname)**
**`vmm_rtl_config_int(name, fname)**
**`vmm_rtl_config_string(name, fname)**
**`vmm_rtl_config_obj(name)**
**`vmm_rtl_config_end(classname)**

Macros for accessing RTL configuration parameters with default implementations.

### Description

Type-specific, shorthand macros providing a default implementation for setting, randomizing, and saving RTL parameter members. The *name* is the name of the member in the class. The *fname* is the name of the RTL configuration parameter in the RTL configuration file.

### Example

```
class ahb_master_config extends vmm_rtl_config;
   rand  int addr_width;
   rand  bit mst_enable;
   string kind = "MSTR";
   `vmm_rtl_config_begin(ahb_master_config)
      `vmm_rtl_config_int(addr_width, mst_width)
      `vmm_rtl_config_boolean(mst_enable, mst_enable)
      `vmm_rtl_config_string(kind, kind)
   `vmm_rtl_config_end(ahb_master_config)
endclass
```

## vmm_rtl_config::map_to_name()

Maps the specified name to the object name.

### SystemVerilog

```
function void vmm_rtl_config::map_to_name(string name)
```

### Description

Use the specified name for this instance of the configuration descriptor, instead of the object name, when looking for relevant `vmm_rtl_config` instances in the RTL configuration hierarchy. The specified name is used as the object name in the “VMM RTL Config” namespace. When argument name is passed as caret (^) for any particular configuration descriptor, that configuration descriptor becomes a root object under "VMM RTL Config".

### Example

```
class ahb_master_config extends vmm_rtl_config;
   function new(string name = "", vmm_rtl_config parent =
         null);
      super.new(name, parent);
   endfunction
endclass

class env_config extends vmm_rtl_config;
   rand ahb_master_config mst_cfg;
   function void build_config_ph();
      mst_cfg = new("mst_cfg", this);
   endfunction
endclass

initial begin
   env_config env_cfg = new("env_cfg");
   env_cfg.mst_cfg.map_to_name("env:mst");
```

```
end
```

## vmm_rtl_config::save_config_ph()

Saves the RTL configuration parameters in a file.

### SystemVerilog

```
virtual function void vmm_rtl_config::save_config_ph()
```

### Description

Creates a configuration file that specifies the RTL configuration parameters corresponding to the current value of the class members. The filename may be computed using the value of the `+vmm_rtl_config` option, using the `vmm_opts::get_string("rtl_config")` method and the hierarchical name of this `vmm_object` instance.

A default implementation of this method is created, if the `` `vmm_rtl_config_*() `` shorthand macros are used.

# vmm_rtl_config_file_format

Base class for RTL configuration file format.

## SystemVerilog

```
virtual class vmm_rtl_config_file_format
```

## Description

This is the base class for RTL configuration file writer or parser. May be used to simplify the task of implementing the `vmm_rtl_config::get_config_ph()` and `vmm_rtl_config::save_config_ph()` methods.

## Example

```
class rtl_config_file_format extends
vmm_rtl_config_file_format;
   virtual function bit fopen(vmm_rtl_config cfg,
         string mode,string fname = "",int lineno = 0);
      string filename = {cfg.prefix, ":",
      cfg.get_object_hiername(), ".rtl_conf"};
      vmm_rtl_config::file_ptr = $fopen(filename, mode);
      if (vmm_rtl_config::file_ptr == 0) return 0;
      else return 1;
   endfunction

   function string get_val(string str);
      if (`vmm_str_match(str, "     : ")) begin
         string fname = `vmm_str_prematch(str);
         string fval = `vmm_str_postmatch(str);
         if (`vmm_str_match(fval, ";")) begin
            fval = `vmm_str_prematch(fval);
         end
         return fval;
      end
```

```
   endfunction

   virtual function bit read_int(string name,
         output int value);
      int r;
      string str;
      $display("Calling read_int for %s", name);
      r = $freadstr(str, vmm_rtl_config::file_ptr);
      str = get_val(str);
      value = str.atoi();
      $display("Got %0d for %s", value, name);
      return (r != 0);
   endfunction

   virtual function bit write_int(string name, int value);
      $fwrite(vmm_rtl_config::file_ptr, "%s : %0d;\n",
         name, value);
      return 1;
   endfunction

   virtual function void fclose();
      $fclose(vmm_rtl_config::file_ptr);
   endfunction

endclass
```

## Summary


- vmm_rtl_config_file_format ::fclose() ..... page B-51
- vmm_rtl_config_file_format::fname() ..... page B-52
- vmm_rtl_config_file_format::fopen() ..... page B-53
- vmm_rtl_config_file_format::get_fname() ..... page B-54
- vmm_rtl_config_file_format::read_bit() ..... page B-55
- vmm_rtl_config_file_format::read_int() ..... page B-56
- vmm_rtl_config_file_format::read_string() ..... page B-57
- vmm_rtl_config_file_format::write_bit() ..... page B-58
- vmm_rtl_config_file_format::write_int() ..... page B-59
- vmm_rtl_config_file_format::write_string() ..... page B-60

## vmm_rtl_config_file_format ::fclose()

Closes the RTL configuration file.

### SystemVerilog

```
pure virtual function void vmm_rtl_config_file_format
   ::fclose()
```

### Description

Closes the configuration file that was previously opened. An implementation may choose to internally cache the information written to the file using the `write_*()` methods, and physically write the file just before closing it.

### Example

```
class rtl_config_file_format extends
      vmm_rtl_config_file_format;
   ...
   virtual function void fclose();
      $fclose(vmm_rtl_config::Xfile_ptrX);
   endfunction
   ...
endclass
```

## vmm_rtl_config_file_format::fname()

Computes the filename that contains the RTL configuration parameter for the specified instance of the RTL configuration descriptor.

### SystemVerilog

```
virtual protected function string
   vmm_rtl_config_file_format::fname(vmm_rtl_config cfg)
```

### Description

Computes the filename that contains the RTL configuration parameter for the specified instance of the RTL configuration descriptor. By default, concatenates the value of the `+vmm_rtl_config` option and the hierarchical name of the specified RTL configuration descriptor, separating the two parts with a slash (/) and appending a `.cfg` suffix.

## vmm_rtl_config_file_format::fopen()

Opens an RTL config file.

### SystemVerilog

```
pure virtual function bit vmm_rtl_config_file_format ::
   fopen(vmm_rtl_config cfg, string mode,
   string fname = "", int lineno = 0)
```

### Description

Opens the configuration file corresponding to the specified RTL configuration descriptor in the specified mode (`r` or `w`). The filename may be computed using the value of the `+vmm_rtl_config` option, using the `vmm_opts::get_string("rtl_config")` method and the name of specified RTL configuration descriptor. Returns true, if the file was successfully opened. If the file is open for read, it may be immediately parsed and its content internally cached. The `fname` and `lineno` arguments are used to track the file name and the line number where `get_config` is invoked from.

### Example

```
class rtl_config_file_format extends
   vmm_rtl_config_file_format;
   virtual function bit fopen(vmm_rtl_config cfg,
         string mode,string fname = "", int lineno = 0);
      string filename = {cfg.prefix, ":",
      cfg.get_object_hiername(), ".rtl_conf"};
      vmm_rtl_config::Xfile_ptrX  = $fopen(filename, mode);
      if (vmm_rtl_config::file_ptr == 0) return 0;
      else return 1;
   endfunction
   ...
endclass
```

## vmm_rtl_config_file_format::get_fname()

Returns the name of the configuration file, which is currently opened. Returns "", if the file is not opened.

### SystemVerilog

```
pure virtual function string vmm_rtl_config_file_format
   ::get_fname()
```

### Description

Returns the name of the configuration file, which is currently opened. Return "", if the file is not opened.

## vmm_rtl_config_file_format::read_bit()

Reads a boolean variable from the RTL configuration file.

### SystemVerilog

```
pure virtual function bit vmm_rtl_config_file_format
   ::read_bit(string name, output bit value)
```

### Description

Returns a boolean value with the specified name, from the RTL configuration file.

### Example

```
class rtl_config_file_format extends
vmm_rtl_config_file_format;
   ...
   virtual function bit read_bit(string name,
         output bit value);
      int r;
      string str;
      r = $freadstr(str, vmm_rtl_config::Xfile_ptrX);
      str = get_val(str);
      value = str.atoi();
      $display("Got %b for %s", value, name);
      return (r != 0);
   endfunction
   ...
endclass
```

## vmm_rtl_config_file_format::read_int()

Reads an integer variable from the RTL configuration file.

### SystemVerilog

```
pure virtual function bit vmm_rtl_config_file_format
   ::read_int(string name, output int value)
```

### Description

Returns an integer value with the specified name, from the RTL configuration file.

### Example

```
class rtl_config_file_format extends
vmm_rtl_config_file_format;
   ...
   virtual function bit read_int(string name,
         output int value);
      int r;
      string str;
      $display("Calling read_int for %s", name);
      r = $freadstr(str, vmm_rtl_config::Xfile_ptrX);
      str = get_val(str);
      value = str.atoi();
      $display("Got %0d for %s", value, name);
      return (r != 0);
   endfunction
   ...
endclass
```

## vmm_rtl_config_file_format::read_string()

Returns a string value with the specified name, from the RTL configuration file.

### SystemVerilog

```
pure virtual function bit vmm_rtl_config_file_format
   ::read_string(string name, output string value)
```

### Description

Sets the *value* argument to the value of the named RTL configuration parameter, as specified in the file. Returns true, if a value for the parameter was found in the file. Otherwise, it returns false. An implementation may require that the parameters be read in the same order, as they are found in the file.

### Example

```
class rtl_config_file_format extends
vmm_rtl_config_file_format;
   ...
   virtual function bit read_string(string name,
         output string value);
      int r;
      string str;
      $display("Calling read_string for %s", name);
      r = $freadstr(str, vmm_rtl_config::Xfile_ptrX);
      value = get_val(str);
      $display("Got %s for %s", value, name);
      return (r != 0);
   endfunction
   ...
endclass
```

## vmm_rtl_config_file_format::write_bit()

Writes a boolean name and value to the RTL config file.

### SystemVerilog

```
pure virtual function bit vmm_rtl_config_file_format
   ::write_bit(string name, bit value)
```

### Description

Writes a name and boolean value to the RTL configuration file. Returns true, if the parameter was not previously written. Otherwise, it returns false. An implementation may physically write the parameter values in the file in a different order, than if they were written using these methods.

### Example

```
class rtl_config_file_format extends
      vmm_rtl_config_file_format;
   ...
   virtual function bit write_bit(string name, bit value);
      $fwrite(vmm_rtl_config::Xfile_ptrX, "%s : %b;\n",
         name, value);
      return 1;
   endfunction
   ...
endclass
```

## vmm_rtl_config_file_format::write_int()

Writes an integer name and value to the RTL config file.

### SystemVerilog

```
pure virtual function bit vmm_rtl_config_file_format
::write_int(string name, int value)
```

### Description

Writes the name and integer value in the RTL configuration file. Returns true, if the parameter was not previously written. Otherwise, it returns false. An implementation may physically write the parameter values in the file in a different order, than if they were written using these methods.

### Example

```
class rtl_config_file_format extends
      vmm_rtl_config_file_format;
   ...
   virtual function bit write_int(string name, int value);
      $fwrite(vmm_rtl_config::Xfile_ptrX, "%s : %0d;\n",
         name, value);
      return 1;
   endfunction
   ...
endclass
```

## vmm_rtl_config_file_format::write_string()

Writes the specified value for the named RTL configuration parameter.

### SystemVerilog

```
pure virtual function bit vmm_rtl_config_file_format
   ::write_string(string name, string value)
```

### Description

Writes the specified value for the named RTL configuration parameter. Returns true, if the parameter was not previously written. Otherwise, it returns false. An implementation may physically write the parameter values in the file in a different order, than if they were written using these methods.

### Example

```
class rtl_config_file_format extends
      vmm_rtl_config_file_format;
   ...
   virtual function bit write_string(string name,
         string value);
      $fwrite(vmm_rtl_config::Xfile_ptrX, "%s : %s;\n",
            name, value);
      return 1;
   endfunction
   ...
endclass
```

# vmm_scenario

Base class for all user-defined scenarios. This class extends from `vmm_data`.

## Summary


- vmm_scenario::get_parent_scenario() ..... page B-62
- vmm_scenario::define_scenario() ..... page B-64
- vmm_scenario::length ..... page B-66
- vmm_scenario::psdisplay() ..... page B-67
- vmm_scenario::redefine_scenario() ..... page B-68
- vmm_scenario::repeat_thresh ..... page B-69
- vmm_scenario::repeated ..... page B-70
- vmm_scenario::repetition ..... page B-72
- vmm_scenario::scenario_id ..... page B-73
- vmm_scenario::scenario_kind ..... page B-74
- vmm_scenario::scenario_name() ..... page B-75
- vmm_scenario::set_parent_scenario() ..... page B-76
- vmm_scenario::stream_id ..... page B-77
- `vmm_scenario_new() ..... page B-78
- `vmm_scenario_member_begin() ..... page B-80
- `vmm_scenario_member_end() ..... page B-82
- `vmm_scenario_member_enum*() ..... page B-83
- `vmm_scenario_member_handle*() ..... page B-85
- `vmm_scenario_member_scalar*() ..... page B-87
- `vmm_scenario_member_string*() ..... page B-89
- `vmm_scenario_member_vmm_data*() ..... page B-91
- `vmm_scenario_member_user_defined() ..... page B-93
- `vmm_scenario_member_vmm_scenario() ..... page B-94

## vmm_scenario::get_parent_scenario()

Returns the higher-level hierarchical scenario.

### SystemVerilog

```
function vmm_scenario get_parent_scenario()
```

### OpenVera

Not supported.

### Description

Returns the single stream or multiple-stream scenario that was specified as the parent of this scenario. A scenario with no parent is a top-level scenario.

### Example

*Example B-1*

```
class atm_cell extends vmm_data;
   ...
endclass

`vmm_scenario_gen(atm_cell, "atm trans")

program test_scenario;
   ...
   atm_cell_scenario parent_scen = new;
   atm_cell_scenario child_scen = new;
   ...
   initial begin
      ...
     vmm_log(log,"Setting parent to a child scenarion \n");
      child.scen.set_parent_scenario(parent_scen);
```

```
      ...
      if(child_scen.get_parent_scenario() == parent_scen)
        vmm_log(log,"Child scenario has proper parent \n");
          ...
      else
       vmm_log(log,"Child scenario has improper parent \n");
      ...
   end
endprogram
```

## vmm_scenario::define_scenario()

Defines a new scenario kind.

### SystemVerilog

```
function int unsigned define_scenario(string name,
    int unsigned max_len=0);
```

### OpenVera

Not supported.

### Description

Defines a new scenario kind that is included in this scenario descriptor, and returns a unique scenario kind identifier. The "vmm_scenario::scenario_kind" data member randomly selects one of the defined scenario kinds. The new scenario kind may contain up to the specified number of random transactions.

The scenario kind identifier should be stored in a state variable that can then be subsequently used to specify the kind-specific constraints.

### Example

*Example B-2*

```
`vmm_scenario_gen(atm_cell, "atm trans")

class my_scenario extends atm_cell_scenario;
   int unsigned START_UP_SEQ;
   int unsigned RESET_SEQ;
   ...
   function new()
```

```
      START_UP_SEQ = define_scenario("START_UP_SEQ",5);
      RESET_SEQ = define_scenario("RESET_SEQ",11);
      ...
   endfunction
   ...
endclass
```

## vmm_scenario::length

Length of the scenario.

### SystemVerilog

```
rand int unsigned length
```

### OpenVera

Not supported.

### Description

Random number of transaction descriptor in this random scenario. Constrained to be less than or equal to the maximum number of transactions in the selected scenario kind.

### Example

*Example B-3*

```
`vmm_scenario_gen(atm_cell, "atm trans")

class my_scenario extends atm_cell_scenario;
   ...
   constraint scen_length {
      if (scenario_kind == START_UP_SEQ)
         { length == 2 } ;
      ... }
endclass
```

## vmm_scenario::psdisplay()

Creates an image of the scenario descriptor.

### SystemVerilog

```
virtual function string psdisplay(string prefix = "")
```

### OpenVera

Not supported.

### Description

Creates human-readable image of the content of the scenario descriptor.

### Example

*Example B-4*

```
class my_scenario extends atm_cell_scenario;
   int unsigned START_UP_SEQ;
     function new()
     redefine_scenario(this.START_UP_SEQ,"WAKE_UP_SEQ",5);
      ...
   endfunction
   ...
endclass
initial begin
   ...
   my_scenario scen_inst = new();
   ...
   $display("Data of the redefined scenario is %s \n",
      scen_inst.psdisplay());
   ...
end
```

## vmm_scenario::redefine_scenario()

Redefines an existing scenario kind.

### SystemVerilog

```
function void redefine_scenario(int unsigned scenario_kind,
    string name, int unsigned max_len=0);
```

### OpenVera

Not supported.

### Description

Redefines an existing scenario kind, which is included in this scenario descriptor. The scenario kind may be redefined with a different name, or maximum number of random transactions.

Use this method to modify, refine, or replace an existing scenario kind, in a pre-defined scenario descriptor.

### Example

*Example B-5*

```
class my_scenario extends atm_cell_scenario;
   int unsigned START_UP_SEQ;
   ...
   function new()
     redefine_scenario(this.START_UP_SEQ,"WAKE_UP_SEQ",5);
       ...
   endfunction
   ...
endclass
```

## vmm_scenario::repeat_thresh

Repetition warning threshold.

### SystemVerilog

```
static int unsigned repeat_thresh
```

### OpenVera

Not supported.

### Description

Specifies a threshold value that triggers a warning about possibly unconstrained "vmm_scenario::repeated" data member. Defaults to 100.

### Example

*Example B-6*

```
`vmm_scenario_gen(atm_cell, "atm trans")

class my_scenario extends atm_cell_scenario;
   ...
   constraint scen_rep_thresh
      {
      if (scenario_kind == START_UP_SEQ)
      { //Note: Default constraint is 100 for repeat_thresh.
            repeat_thresh < 120 } ;
      ...
      }
endclass
```

## vmm_scenario::repeated

Scenario identifier of the randomizing generator.

### SystemVerilog

```
rand int unsigned repeated
```

### OpenVera

Not supported.

### Description

The number of time the entire scenario is repeated. A repetition value of zero specifies that the scenario will not be repeated, and will be applied only once.

Constrained to zero, by default, by the "vmm_scenario::repetition" constraint block.

Note:It is best to repeat the same transaction, instead of creating a scenario of many transactions constrained to be identical.

### Example

*Example B-7*

```
`vmm_scenario_gen(atm_cell, "atm trans")

class my_scenario extends atm_cell_scenario;
   ...
   constraint scen_repetitions
      {
      if (scenario_kind == START_UP_SEQ)
         { //Note: Default constraint is 0 for repeated.
```

```
            repeated < 4 } ;
        ...
        }
endclass
```

## vmm_scenario::repetition

Constraint preventing the scenario, from being repeated.

### SystemVerilog

```
constraint repetition {
   repeated == 0;
}
```

### OpenVera

Not supported.

### Description

The "vmm_scenario::repeated" data member specifies the number of times a scenario is repeated. It is not often used, but if left unconstrained, can cause stimulus to be erroneously repeatedly applied over two billion times on an average.

This constraint block constrains this data member to prevent repetition, by default. To have a scenario be repeated a random number of times, override this constraint block.

### Example

*Example B-8*

```
class many_atomic_scenario
   extends eth_frame_atomic_scenario;
   constraint repetition {repeated < 10;}
endclass
```

## vmm_scenario::scenario_id

Scenario identifier of the randomizing generator.

### SystemVerilog

```
int scenario_id
```

### OpenVera

Not supported.

### Description

This data member is set by the scenario generator, before randomization to the current scenario counter value of the generator. This state variable can be used to specifiy scenario-specific constraints, or to identify the order of different scenarios within a stream.

### Example

*Example B-9*

```
class atm_cell extends vmm_data;
   rand int payload[3];
   ...
endclass
`vmm_scenario_gen(atm_cell, "atm trans")
class atm_cell_ext extends atm_cell;
   ...
   constraint test {
      payload[1] == scenario_id;
      ...
   }
endclass
```

## vmm_scenario::scenario_kind

Scenario kind identified.

### SystemVerilog

```
rand int unsigned scenario_kind
```

### OpenVera

Not supported.

### Description

Used to randomly select one of the scenario kinds, which is defined in this random scenario descriptor.

### Example

*Example B-10*

```
`vmm_scenario_gen(atm_cell, "atm trans")

class my_scenario extends atm_cell_scenario;
   ...
   constraint start_up_const {
      (trans_type == 0 ) -> {scenario_kind inside
         {RESET_SEQ,START_UP_SEQ}};
      ...
   }
endclass
```

## vmm_scenario::scenario_name()

Returns the name of a scenario kind.

### SystemVerilog

```
function string scenario_name(int unsigned scenario_kind);
```

### OpenVera

Not supported.

### Description

Returns the name of the specified scenario kind, as defined by the "vmm_scenario::define_scenario()" or "vmm_scenario::redefine_scenario()" methods.

### Example

*Example B-11*

```
class my_scenario extends atm_cell_scenario;
   int unsigned START_UP_SEQ;
   ...
   function new()
     redefine_scenario(this.START_UP_SEQ,"WAKE_UP_SEQ",5);
      ...
   endfunction
   ...
   function post_randomize();
      $display("Name of the redefined scenario is %s \n",
         scenario_name(scenario_kind));
      ...
   endfunction
endclass
```

## vmm_scenario::set_parent_scenario()

Defines higher-level hierarchical scenario.

### SystemVerilog

```
function void set_parent_scenario(
    vmm_scenario parent)
```

### OpenVera

Not supported.

### Description

Specifies the single stream or multiple-stream scenario that is the parent of this scenario. This allows this scenario to grab a channel that is already grabbed by the parent scenario.

### Example

*Example B-12*

```
class atm_cell extends vmm_data;
   rand int payload[3];
endclass

`vmm_scenario_gen(atm_cell, "atm trans")

program test_scenario;
   atm_cell_scenario parent_scen = new;
   atm_cell_scenario child_scen = new;
   initial begin
     vmm_log(log,"Setting parent to a child scenarion \n");
        child.scen.set_parent_scenario(parent_scen);
   end
endprogram
```

## vmm_scenario::stream_id

Stream identifier of the randomizing generator.

### SystemVerilog

```
int stream_id
```

### OpenVera

Not supported.

### Description

This data member is set by the scenario generator, before randomization to the stream identifier of generator. This state variable can be used to specific stream-specific constraints, or to differentiate stimulus from different streams in a scoreboard.

### Example

*Example B-13*

```
class atm_cell extends vmm_data;
   rand int payload[3];
   ...
endclass

`vmm_scenario_gen(atm_cell, "atm trans")

class atm_cell_ext extends atm_cell;
   ...
   constraint test {
      payload[0] == stream_id;
      ...}
endclass
```

## ‘vmm_scenario_new()

Start of explicit constructor implementation.

### SystemVerilog

```
‘vmm_scenario_new(class-name)
```

### OpenVera

Not supported.

### Description

Specifies that an explicit user-defined constructor is used, instead of the default constructor provided by the shorthand macros. Also, declares a “vmm_log” instance that can be passed to the base class constructor. Use this macro when data members must be explicitly initialized in the constructor.

The class-name specified must be the name of the vmm_scenario extension class that is being implemented.

This macro should be followed by the constructor declaration, and must precede the shorthand data member section. This means that it should be located before the “‘vmm_scenario_member_begin()” macro.

### Example

*Example B-14*

```
class my_scenario extends vmm_ms_scenario;
   ...
   ‘vmm_scenario_new(my_scenario)
```

```
      function new(vmm_scenario parent = null);
         super.new(parent)
         ...
      endfunction

   `vmm_scenario_member_begin(my_scenario)
      ...
   `vmm_scenario_member_end(my_scenario)
   ...
endclass
```

## ‘vmm_scenario_member_begin()

Start of shorthand section.

### SystemVerilog

```
‘vmm_scenario_member_begin(class-name)
```

### OpenVera

Not supported.

### Description

Starts the shorthand section providing a default implementation for the `psdisplay()`, `is_valid()`, `allocate()`, `copy()`, and `compare()` methods. A default implementation for the constructor is also provided unless the “‘vmm_scenario_new()” macro as been previously specified.

The class-name specified must be the name of the `vmm_scenario` extension class that is being implemented.

The shorthand section can only contain shorthand macros and must be terminated by the “‘vmm_scenario_member_end()” method.

### Example

*Example B-15*

```
class my_scenario extends vmm_data;
   ...
   ‘vmm_scenario_member_begin(my_scenario)
      ...
   ‘vmm_scenario_member_end(my_scenario)
```

```
endclass
```

## ‘vmm_scenario_member_end()

End of shorthand section.

### SystemVerilog

```
‘vmm_scenario_member_end(class-name)
```

### OpenVera

Not supported.

### Description

Terminates the shorthand section, by providing a default implementation for the `psdisplay()`, `is_valid()`, `allocate()`, `copy()`, and `compare()` methods.

The class-name specified must be the name of the `vmm_scenario` extension class that is being implemented.

The shorthand section must be started by the “`‘vmm_scenario_member_begin()`” method.

### Example

*Example B-16*

```
class eth_scenario extends vmm_data;
   ...
   ‘vmm_scenario_member_begin(eth_scenario)
      ...
   ‘vmm_scenario_member_end(eth_scenario)
   ...
endclass
```

## ‘vmm_scenario_member_enum*()

The shorthand implementation for an enumerated data member.

### SystemVerilog

```
‘vmm_scenario_member_enum(member-name,
                          vmm_data::do_what_e do_what)

‘vmm_scenario_member_enum_array(member-name,
                           vmm_data::do_what_e do_what)

‘vmm_scenario_member_enum_da(member-name,
                           vmm_data::do_what_e do_what)

‘vmm_scenario_member_enum_aa_scalar(member-name,
                              vmm_data::do_what_e do_what)

‘vmm_scenario_member_enum_aa_string(member-name,
                              vmm_data::do_what_e do_what)
```

### OpenVera

Not supported.

### Description

Adds the specified enum-type, fixed array of enums, dynamic array of enums, scalar-indexed associative array of enums, or string-indexed associative array of enums data member to the default implementation of the methods that are specified by the `do_what` argument.

The shorthand implementation must be located in a section started by “‘vmm_scenario_member_begin()” .

## Example

*Example B-17*

```
typedef enum bit[1:0] {NORMAL, VLAN, JUMBO } frame_type;

class eth_scenario extends vmm_data;
   rand frame_type frame_var;
   ...
   `vmm_scenario_member_begin(eth_scenario)
      `vmm_scenario_member_enum(frame_var, DO_ALL)
      ...
   `vmm_scenario_member_end(eth_scenario)
   ...
endclass
```

## ‘vmm_scenario_member_handle*()

The shorthand implementation for a class handle data member.

### SystemVerilog

```
‘vmm_scenario_member_handle(member-name,
                            vmm_data::do_what_e do_what)

‘vmm_scenario_member_handle_array(member-name,
                            vmm_data::do_what_e do_what)

‘vmm_scenario_member_handle_da(member-name,
                            vmm_data::do_what_e do_what)

‘vmm_scenario_member_handle_aa_scalar(member-name,
                            vmm_data::do_what_e do_what)

‘vmm_scenario_member_handle_aa_string(member-name,
                            vmm_data::do_what_e do_what)
```

### OpenVera

Not supported.

### Description

Adds the specified handle-type fixed array of handles, dynamic array of handles, scalar-indexed associative array of handles, or string-indexed associative array of handles data member to the default implementation of the methods that are specified by the `do_what` argument.

The shorthand implementation must be located in a section started by “‘vmm_scenario_member_begin()” .

## Example

*Example B-18*

```
class vlan_frame;
   ...
endclass

class eth_scenario extends vmm_data;
   vlan_frame vlan_fr_var ;
   ...
   `vmm_scenario_member_begin(eth_scenario)
      `vmm_scenario_member_vmm_handle(vlan_fr_var,
        DO_ALL,DO_DEEP)
      ...
   `vmm_scenario_member_end(eth_scenario)
   ...
endclass
```

## ‘vmm_scenario_member_scalar*()

The shorthand implementation for a scalar data member.

### SystemVerilog

```
‘vmm_scenario_member_scalar(member-name,
                            vmm_data::do_what_e do_what)

‘vmm_scenario_member_scalar_array(member-name,
                            vmm_data::do_what_e do_what)

‘vmm_scenario_member_scalar_da(member-name,
                            vmm_data::do_what_e do_what)

‘vmm_scenario_member_scalar_aa_scalar(member-name,
                            vmm_data::do_what_e do_what)

‘vmm_scenario_member_scalar_aa_string(member-name,
                            vmm_data::do_what_e do_what)
```

### OpenVera

Not supported.

### Description

Adds the specified scalar-type, fixed array of scalars, dynamic array of scalars, scalar-indexed associative array of scalars, or string-indexed associative array of scalars data member to the default implementation of the methods that are specified by the `do_what` argument.

A scalar is an integral type, such as bit, bit vector, and packed unions.

The shorthand implementation must be located in a section started by “‘vmm_scenario_member_begin()” .

## Example

*Example B-19*

```
class eth_scenario extends vmm_data;
   rand bit [47:0] da;
   ...
   ‘vmm_scenario_member_begin(eth_scenario)
      ‘vmm_scenario_member_scalar(da, DO_ALL);
      ...
   ‘vmm_scenario_member_end(eth_scenario)
   ...
endclass
```

## ‘vmm_scenario_member_string*()

The shorthand implementation for a string data member.

### SystemVerilog

```
‘vmm_scenario_member_string(member-name,
                            vmm_data::do_what_e do_what)

‘vmm_scenario_member_string_array(member-name,
                            vmm_data::do_what_e do_what)

‘vmm_scenario_member_string_da(member-name,
                            vmm_data::do_what_e do_what)

‘vmm_scenario_member_string_aa_scalar(member-name,
                            vmm_data::do_what_e do_what)

‘vmm_scenario_member_string_aa_string(member-name,
                            vmm_data::do_what_e do_what)
```

### OpenVera

Not supported.

### Description

Adds the specified string-type, fixed array of strings, dynamic array of strings, scalar-indexed associative array of strings, or string-indexed associative array of strings data member to the default implementation of the methods that are specified by the `do_what` argument.

The shorthand implementation must be located in a section started by “‘vmm_scenario_member_begin()” .

## Example

*Example B-20*

```
class eth_scenario extends vmm_data;
   string scen_name;
   ...
   `vmm_scenario_member_begin(eth_scenario)
      `vmm_scenario_member_string(scen_name, DO_ALL)
      ...
   `vmm_scenario_member_end(eth_scenario)
   ...
endclass
```

## ‘vmm_scenario_member_vmm_data*()

The shorthand implementation for a vmm_data-based data member.

### SystemVerilog

```
‘vmm_scenario_member_vmm_data(member-name,
                              vmm_data::do_what_e do_what,
                             vmm_data::do_how_e  do_how)

‘vmm_scenario_member_vmm_data_array(member-name,
                              vmm_data::do_what_e do_what,
                              vmm_data::do_how_e do_how)

‘vmm_scenario_member_vmm_data_da(member-name,
                              vmm_data::do_what_e do_what,
                              vmm_data::do_how_e do_how)

‘vmm_scenario_member_vmm_data_aa_scalar(member-name,
                              vmm_data::do_what_e do_what,
                              vmm_data::do_how_e do_how)

‘vmm_scenario_member_vmm_data_aa_string(member-name,
                              vmm_data::do_what_e do_what,
                              vmm_data::do_how_e do_how)
```

### OpenVera

Not supported.

### Description

Adds the specified vmm_data-type, fixed array of vmm_datas, dynamic array of vmm_datas, scalar-indexed associative array of vmm_datas, or string-indexed associative array of vmm_datas data member to the default implementation of the methods that are

specified by the `do_what` argument. The `do_how` argument specifies whether the `vmm_data` values must be processed deeply or shallowly.

The shorthand implementation must be located in a section started by "`vmm_scenario_member_begin()" .

## Example

*Example B-21*

```
class vlan_frame extends vmm_data;
   ...
endclass

class eth_scenario extends vmm_data;
   vlan_frame vlan_fr_var ;
   ...
   `vmm_scenario_member_begin(eth_scenario)
      `vmm_scenario_member_vmm_data(vlan_fr_var,
          DO_ALL,DO_DEEP)
      ...
   `vmm_scenario_member_end(eth_scenario)
   ...
endclass
```

## ‘vmm_scenario_member_user_defined()

User-defined shorthand implementation data member.

### SystemVerilog

```
‘vmm_scenario_member_user_defined(member-name,
                                  vmm_data::do_what_e do_what)
```

### OpenVera

Not supported.

### Description

Adds the specified user-defined default implementation of the methods that are specified by the `do_what` argument.

The shorthand implementation must be located in a section started by “‘vmm_scenario_member_begin()” .

### Example

*Example B-22*

```
class eth_scenario extends vmm_data;
   rand bit[47:0] da;
   `vmm_scenario_member_begin(eth_scenario)
      `vmm_scenario_member_user_defined(da, DO_ALL)
   `vmm_scenario_member_end(eth_scenario)

   function bit do_da ( input vmm_data::do_what_e do_what)
      do_da = 1;  // Success, abort by returning 0
      case (do_what)
      endcase
   endfunction
endclass
```

## ‘vmm_scenario_member_vmm_scenario()

The shorthand implementation for a sub-scenario.

### SystemVerilog

```
‘vmm_scenario_member_vmm_scenario(member-name,
                                  vmm_data::do_what_e do_what)
```

### OpenVera

Not supported.

### Description

Adds the specified vmm_scenario-type sub-scenario member to the default implementation of the methods that are specified by the `do_what` argument.

The shorthand implementation must be located in a section started by “‘vmm_scenario_member_begin()” .

### Example

*Example B-23*

```
class vlan_scenario extends vmm_data;
   ...
endclass

class eth_scenario extends vmm_data;
   vlan_scenario vlan_scen ;
   `vmm_scenario_member_begin(eth_scenario)
      `vmm_scenario_member_vmm_scenario(vlan_scen,
         DO_ALL)
   `vmm_scenario_member_end(eth_scenario)
endclass
```

# vmm_scenario_gen#(T, text)

Parameterized version of the VMM scenario generator.

## SystemVerilog

```
class vmm_scenario_gen #(type T=vmm_data,string text= “”)
extends vmm_scenario_gen_base;
```

## Description

The `vmm_scenario_generator macro creates a parameterized scenario generator. This generator can generate non-vmm_data transactions as well.

A macro is used to define a *class-name*_**scenario_gen** class, for any user-specified class derived from **vmm_data**[1], using a process similar to the **‘vmm_channel** macro.

The scenario generator class is an extension of the **vmm_xactor** class and as such, inherits all the public interface elements provided in the base class.

## Example

```
class ahb_trans extends vmm_data;
   rand bit [31:0] addr;
   rand bit [31:0] data;
endclass

`vmm_channel(ahb_trans)
`vmm_scenario_gen(ahb_trans, "AHB Scenario Gen")
```

1. With a constructor callable without any arguments.

```
ahb_trans_channel chan0 = new("ahb_trans_chan", "chan0");
ahb_trans_scenario_gen gen1  = new("AhbGen1", 0, chan0);
```

Is the same as:

```
vmm_channel_typed#(ahb_trans) chan0 = new("ahb_trans_chan",
   "chan0");
vmm_scenario_gen #(ahb_trans, AHB Scenario Gen") gen1 =
new("AhbGen1", 0, chan0);
```

## Summary


- vmm_scenario_gen::define_scenario() ..... page B-97
- vmm_scenario_gen::enum {DONE} ..... page B-98
- vmm_scenario_gen::enum {GENERATED} ..... page B-100
- vmm_scenario_gen::get_all_scenario_names() ..... page B-102
- vmm_scenario_gen::get_n_insts() ..... page B-103
- vmm_scenario_gen::get_n_scenarios() ..... page B-104
- vmm_scenario_gen::get_names_by_scenario() ..... page B-105
- vmm_scenario_gen::get_scenario() ..... page B-106
- vmm_scenario_gen::get_scenario_index() ..... page B-107
- vmm_scenario_gen::get_scenario_name() ..... page B-109
- vmm_scenario_gen::inject() ..... page B-110
- vmm_scenario_gen::inject_obj() ..... page B-112
- vmm_scenario_gen::inst_count ..... page B-114
- vmm_scenario_gen::new() ..... page B-115
- vmm_scenario_gen::out_chan ..... page B-117
- vmm_scenario_gen::replace_scenario() ..... page B-118
- vmm_scenario_gen::register_scenario() ..... page B-120
- vmm_scenario_gen::scenario_count ..... page B-122
- vmm_scenario_gen::scenario_exists() ..... page B-123
- vmm_scenario_gen::scenario_set[$] ..... page B-125
- vmm_scenario_gen::select_scenario ..... page B-127
- vmm_scenario_gen::stop_after_n_insts ..... page B-129
- vmm_scenario_gen::stop_after_n_scenarios ..... page B-131
- vmm_scenario_gen::unregister_scenario() ..... page B-133
- vmm_scenario_gen::unregister_scenario_by_name() .. page B-134
- `vmm_scenario_gen ..... page B-136
- `vmm_scenario_gen_using() ..... page B-138

## vmm_scenario_gen::define_scenario()

Defines a new scenario kind.

### SystemVerilog

```
function int unsigned define_scenario(string name,
    int unsigned max-len);
```

### OpenVera

Not supported.

### Description

Defines a new scenario kind that is included in this scenario descriptor, and returns a unique scenario kind identifier. The "vmm_scenario::scenario_kind" data member randomly selects one of the defined scenario kinds. The new scenario kind may contain up to the specified number of random transactions.

The scenario kind identifier should be stored in a state variable that can then be subsequently used to the specified kind-specific constraints.

## vmm_scenario_gen::enum {DONE}

Notification identifier for the **`vmm_xactor::notify`** notification service interface.

### SystemVerilog

```
enum {DONE};
```

### OpenVera

Not supported.

### Description

Notification identifier for the **`vmm_xactor::notify`** notification service interface provided by the **`vmm_xactor`** base class. It is configured as a **`vmm_notify::ON_OFF`** notification, and is indicated when the generator stops, because the specified number of instances or scenarios are generated. No status information is specified.

### Example

*Example B-24*

```
program test_scenario;
   ...
   atm_cell_scenario_gen atm_gen =
      new("Atm Scenario Gen", 12);
   ...
   initial
   begin
      ...
      atm_gen.stop_after_n_scenarios = 10;
      atm_gen.start_xactor();
```

```
      ...
     atm_gen.notify.wait_for(atm_cell_scenario_gen::DONE);
    $finish;
  end
  ...
endprogram
```

## vmm_scenario_gen::enum {GENERATED}

Notification identifier for the **`vmm_xactor::notify`** notification service interface.

### SystemVerilog

```
enum {GENERATED};
```

### OpenVera

Not supported.

### Description

Notification identifier for the **`vmm_xactor::notify`** notification service interface provided by the **`vmm_xactor`** base class. It is configured as a **`vmm_notify::ONE_SHOT`** notification, and is indicated immediately before a scenario is applied to the output channel. The randomized scenario is specified as the status of the notification.

### Example

*Example B-25*

```
program test_scenario;
   ...
   atm_cell_scenario_gen atm_gen =
      new("Atm Scenario Gen",  12);
   ...
   initial
   begin
      ...
      atm_gen.stop_after_n_scenarios = 10;
      atm_gen.start_xactor();
```

```
      ...
      atm_gen.notify.wait_for(
         atm_cell_scenario_gen::GENERATED);
   end
   ...
endprogram
```

## vmm_scenario_gen::get_all_scenario_names()

Returns all names in the scenario registry.

### SystemVerilog

```
virtual function void get_all_scenario_names(
    ref string    name[$])
```

### OpenVera

Not supported.

### Description

Appends the names under which a scenario descriptor is registered. Returns the number of names that were added to the array.

### Example

*Example B-26*

```
class atm_cell extends vmm_data;
   ...
endclass
`vmm_scenario_gen(atm_cell, "atm trans")
program test_scenario;
   string scen_names_arr[$];
   atm_cell_scenario_gen atm_gen =
      new("Atm Scenario Gen", 12);
   atm_cell_scenario atm_scenario = new;
   ...
   initial begin
      ...
      atm_gen.get_all_scenario_names(scen_names_arr);
   end
endprogram
```

## vmm_scenario_gen::get_n_insts()

Returns the actual number of instances generated.

### SystemVerilog

```
function int unsigned get_n_insts();
```

### OpenVera

Not supported.

### Description

The generator stops after the **`stop_after_n_insts`** limit on the number of instances is reached, and only after entire scenarios are applied. Hence, it can generate a few more instances than configured. This method returns the actual number of instances that were generated.

### Example

*Example B-27*

```
program test_scenario;
   atm_cell_scenario_gen atm_gen =
      new("Atm Scenario Gen", 12);
   initial
   begin
      atm_gen.stop_after_n_insts = 10;
      atm_gen.start_xactor();
      `vmm_note(log,$psprintf(
         "Total Instances Generated: %0d",
         atm_gen.get_n_insts()));
   end
endprogram
```

## vmm_scenario_gen::get_n_scenarios()

Returns the actual number of scenarios generated.

### SystemVerilog

```
function int unsigned get_n_scenarios();
```

### OpenVera

Not supported.

### Description

The generator stops after the **`stop_after_n_scenarios`** limit on the number of scenarios is reached, and only after entire scenarios are applied. Hence, it can generate a few less scenarios than configured. This method returns the actual number of scenarios that were generated.

### Example

*Example B-28*

```
program test_scenario;
   atm_cell_scenario_gen atm_gen =
      new("Atm Scenario Gen", 12);
   initial
   begin
      atm_gen.stop_after_n_scenarios = 10;
      atm_gen.start_xactor();
      `vmm_note(log,$psprintf("Total Scenarios Generated:
          %0d", atm_gen.get_n_scenarios()));
   end
   ...
endprogram
```

## vmm_scenario_gen::get_names_by_scenario()

Returns the names under which a scenario is registered.

### SystemVerilog

```
virtual function void get_names_by_scenario(
     vmm_ss_scenario_base scenario,ref string name[$])
```

### OpenVera

Not supported.

### Description

Appends the names under which the specified scenario descriptor is registered. Returns the number of names that were added to the array.

### Example

*Example B-29*

```
class atm_cell extends vmm_data;
endclass
`vmm_scenario_gen(atm_cell, "atm trans")
program test_scenario;
   string scen_names_arr[$];
   atm_cell_scenario_gen atm_gen =
      new("Atm Scenario Gen", 12);
   atm_cell_scenario atm_scenario = new;
   initial begin
      atm_gen.get_names_by_scenario(
         atm_scenario,scen_names_arr);
   end
endprogram
```

## vmm_scenario_gen::get_scenario()

Returns the scenario registered under a specified name.

### SystemVerilog

```
virtual function vmm_scenario get_scenario(string name)
```

### OpenVera

Not supported.

### Description

Returns the scenario descriptor registered under the specified *name*. Generates a warning message and returns `NULL`, if there are no scenarios registered under that name.

### Example

*Example B-30*

```
class atm_cell extends vmm_data;
endclass

`vmm_scenario_gen(atm_cell, "atm trans")

program test_scenario;
   atm_cell_scenario_gen atm_gen =
      new("Atm Scenario Gen", 12);
   atm_cell_scenario atm_scenario = new;
   ...
   initial begin
    if(atm_gen.get_scenario("PARENT SCEN") == atm_scenario)
         vmm_log(log,"Scenario matching \n");
   end
endprogram
```

## vmm_scenario_gen::get_scenario_index()

Returns the index of the specified scenario.

### SystemVerilog

```
virtual function int get_scenario_index(
    vmm_ss_scenario_base scenario)
```

### OpenVera

Not supported.

### Description

Returns the index of the specified scenario descriptor, which is in the scenario set array. A warning message is generated and returns -1, if the scenario descriptor is not found in the scenario set.

### Example

*Example B-31*

```
class atm_cell extends vmm_data;
   ...
endclass

`vmm_scenario_gen(atm_cell, "atm trans")

program test_scenario;
   atm_cell_scenario_gen atm_gen =
      new("Atm Scenario Gen", 12);
   atm_cell_scenario atm_scenario = new;
   ...
   initial begin
      ...
    scen_index = atm_gen.get_scenario_index(atm_scenario);
```

```
      if(scen_index == 5)
         `vmm_note(log, `vmm_sformatf(
            "INDEX MATCHED %0d", index));
      else
         `vmm_error(log,`vmm_sformatf(
            "INDEX NOT MATCHING %0d", index));
      ...
   end
endprogram
```

## vmm_scenario_gen::get_scenario_name()

Returns the name of the specified scenario.

### SystemVerilog

```
virtual function int get_scenario_name(vmm_scenario
scenario)
```

### OpenVera

Not supported.

### Description

Returns a name under which the specified scenario descriptor is registered. Returns `""`, if the scenario is not registered.

### Example

*Example B-32*

```
class atm_cell extends vmm_data;
endclass

`vmm_scenario_gen(atm_cell, "atm trans")

program test_scenario;
   atm_cell_scenario_gen atm_gen =
      new("Atm Scenario Gen", 12);
   atm_cell_scenario atm_scenario = new;
   initial begin
      scenario_name =
atm_gen.get_scenario_name(atm_scenario);
      vmm_note(log,`vmm_sformatf("Registered name for
atm_scenario is : %s\n",scenario_name));
   end
endprogram
```

## vmm_scenario_gen::inject()

Injects the specified scenario descriptor in the output stream.

### SystemVerilog

```
virtual task inject(vmm_ss_scenario#(T) scenario);
```

### OpenVera

Not supported.

### Description

Unlike injecting the descriptors directly in the output channel, it counts toward the number of instances and scenarios generated by this generator, and will be subjected to the callback methods. The method returns once the scenario is consumed by the output channel, or it is dropped by the callback methods.

This method can be used to inject directed stimulus while the generator is running (with unpredictable timing), or when the generated is stopped.

### Example

*Example B-33*

```
class my_scenario extends atm_cell_scenario
   ...
   virtual task apply(atm_cell_channel channel,
         ref int unsigned n_insts);
      ...
      this.randomize();
      super.apply(channel, n_insts);
      ...
```

```
   endtask
...
endclass

program test_scenario;
   ...
   atm_cell_scenario_gen atm_gen =
      new("Atm Scenario Gen", 12);
   my_scenario scen;
   ...
   initial
   begin
      ...
      atm_gen.stop_after_n_scenarios = 10;
      atm_gen.start_xactor();
      ...
      atm_gen.inject(scen);
      ...
   end
   ...
endprogram
```

## vmm_scenario_gen::inject_obj()

Injects the specified descriptor in the output stream.

### SystemVerilog

```
virtual task inject_obj(class-name obj);
```

### OpenVera

Not supported.

### Description

Unlike injecting the descriptor directly in the output channel, it counts toward the number of instances and scenarios generated by this generator, and will be subjected to the callback methods as an atomic scenario. The method returns once the descriptor is consumed by the output channel, or it is dropped by the callback methods.

This method can be used to inject directed stimulus while the generator is running (with unpredictable timing), or when the generated is stopped.

### Example

*Example B-34*

```
program test_scenario;
   ...
   atm_cell_scenario_gen atm_gen =
      new("Atm Scenario Gen", 12, genchan);
   atm_cell tr = new();
   ...
```

```
   initial
   begin
      ...
      tr.addr = 64'ha0;
      tr.data = 64'h50;
      atm_gen.stop_after_n_scenarios = 10;
      atm_gen.start_xactor();
      ...
      atm_gen.inject_obj(tr);
      ...
   end
   ...
endprogram
```

## vmm_scenario_gen::inst_count

Returns the number of instances generated so far.

### SystemVerilog

```
protected int inst_count;
```

### OpenVera

```
protected integer inst_count;
```

### Description

Returns the current count of the number of individual instances generated by or injected through the scenario generator. When it reaches or surpasses the value in `vmm_scenario_gen::stop_after_n_insts`, the generator stops.

### Example

*Example B-35*

```
class generator_ext extends pkt_scenario_gen;
   ...
   function void reset_xactor(reset_e rst_typ = SOFT_RST);
      this.inst_count     = 0;
      ...
   endfunction
endclass
```

## vmm_scenario_gen::new()

Creates a new instance of a scenario generator transactor.

### SystemVerilog

```
function new(string instance,
    int stream_id = -1, class-name_channel out_chan =
null,vmm_object parent = null);
```

### OpenVera

Not supported.

### Description

Creates a new instance of a scenario generator transactor, with the specified instance name and optional stream identifier. The generator can be optionally connected to the specified output channel. If no output channel is specified, one will be created internally in the *`class-name`*`_`**`scenario_gen::out_chan`** property.

The name of the transactor is defined as the user-defined class description string, which is specified in the class implementation macro appended with the "*Scenario Generator*". Specified `parent` argument indicates the parent of this generator.

### Example

*Example B-36*

```
program test_scenario;
   ...
   atm_cell_scenario_gen atm_gen =
```

```
      new("Atm Scenario Gen", 12);
endprogram
```

## vmm_scenario_gen::out_chan

References the output channel for the instances generated by this transactor.

### SystemVerilog

```
class-name_channel out_chan;
```

### OpenVera

Not supported.

### Description

The output channel may be specified through the constructor. If no output channel was specified, a new instance is automatically created. The reference in this property may be dynamically replaced, but the generator should be stopped during the replacement.

### Example

*Example B-37*

```
program test_scenario;
   atm_cell_scenario_gen atm_gen =
      new("Atm Scenario Gen", 12);
   initial
   begin
      atm_gen.stop_after_n_insts = 10;
      atm_gen.start_xactor();
      while (1) begin
         atm_gen.out_chan.get(c);
      end
   end
endprogram
```

## vmm_scenario_gen::replace_scenario()

Replaces a scenario descriptor.

### SystemVerilog

```
virtual function void replace_scenario(string name,
    <class-name>_scenario scenario);
```

### OpenVera

Not supported.

### Description

Registers the specified scenario under the specified name, replacing the scenario that is previously registered under that name, if any. The name under which a scenario is registered does not need to be the same as the name of a kind of scenario, which is defined in the scenario descriptor using the `vmm_scenario_gen::define_scenario()` method. The same scenario may be registered multiple times under different names, therefore creating an alias to the same scenario.

Registering a scenario implicitly appends it to the scenario set, if it is not already in the `vmm_scenario_gen::scenario_set[$]` array. The replaced scenario is removed from the scenario set, if it is not also registered under another name.

### Example

*Example B-38*

```
`vmm_scenario_gen(atm_cell, "atm trans")
```

```
program test_scenario;
   atm_cell_scenario_gen atm_gen =
      new("Atm Scenario Gen", 12);
   atm_cell_scenario parent_scen = new;
   ...
   initial begin
     ...
    atm_gen.register_scenario("MY SCENARIO", parent_scen);
    atm_gen.register_scenario("PARENT SCEN", parent_scen);
    ...
    if(atm_gen.scenario_exists("MY SCENARIO")
       begin
          atm_gen.replace_scenario(
             "MY SCENARIO", parent_scen);
          vmm_log(log,
             "Scenario exists and has been replaced\n");
          ...
      end
   end
endprogram
```

## vmm_scenario_gen::register_scenario()

Registers a scenario descriptor.

### SystemVerilog

```
virtual function void register_scenario(string name,
    vmm_ss_scenario_base scenario);
```

### OpenVera

Not supported.

### Description

Registers the specified scenario under the specified name. The name under which a scenario is registered does not need to be the same as the name of a kind of scenario, which is defined in the scenario descriptor using the **vmm_scenario_gen::define_scenario()** method. The same scenario may be registered multiple times under different names, therefore creating an alias to the same scenario.

Registering a scenario implicitly appends it to the scenario set, if it is not already in the **vmm_scenario_gen::scenario_set[$]** array.

It is an error to register a scenario under a name that already exists. Use the **vmm_scenario_gen::replace_scenario()** method to replace a registered scenario.

## Example

*Example B-39*

```
class atm_cell extends vmm_data;
   ...
endclass

`vmm_scenario_gen(atm_cell, "atm trans")

program test_scenario;
   atm_cell_scenario_gen atm_gen =
      new("Atm Scenario Gen", 12);
   atm_cell_scenario parent_scen = new;
   ...
   initial begin
      ...
      vmm_log(log,"Registering scenario \n");
     atm_gen.register_scenario("PARENT SCEN", parent_scen);
      ...
   end
endprogram
```

## vmm_scenario_gen::scenario_count

Returns the number of scenarios generated so far.

### SystemVerilog

```
protected int scenario_count;
```

### OpenVera

```
protected integer scenario_count;
```

### Description

Returns the current count of the number of scenarios generated by or injected through the scenario generator. When it reaches or surpasses the value in `vmm_scenario_gen::stop_after_n_scenarios`, the generator stops.

### Example

*Example B-40*

```
class generator_ext extends pkt_scenario_gen;
   ...
   virtual task inject(pkt_scenario scenario);
      scenario.scenario_id = this.scenario_count;
      ...
   endtask
endclass
```

## vmm_scenario_gen::scenario_exists()

Checks whether a scenario is registered under a specified name or not.

### SystemVerilog

```
virtual function bit scenario_exists(string name)
```

### OpenVera

Not supported.

### Description

Returns `TRUE`, if there is a scenario registered under the specified name. Otherwise, it returns `FALSE`.

Use the **`vmm_scenario_gen::get_scenario()`** method to retrieve a scenario under a specified name.

### Example

*Example B-41*

```
class atm_cell extends vmm_data;
   ...
endclass

`vmm_scenario_gen(atm_cell, "atm trans")

program test_scenario;
   atm_cell_scenario_gen atm_gen =
      new("Atm Scenario Gen", 12);
   atm_cell_scenario parent_scen = new;
   ...
```

```
   initial begin
      ...
      vmm_log(log,"Registering scenario \n");
     atm_gen.register_scenario("PARENT SCEN", parent_scen);
      ...
      if(atm_gen.scenario_exists("PARENT SCEN") begin
        vmm_log(log,"Scenario exists and you can use \n");
         ...
      end
   end
endprogram
```

## vmm_scenario_gen::scenario_set[$]

Sets-of available scenario descriptors that may be repeatedly randomized.

### SystemVerilog

```
vmm_ss_scenario(T) scenario_set[$];
```

### OpenVera

Not supported.

### Description

Sets-of available scenario descriptors that may be repeatedly randomized, to create the random content of the output stream. The *class-name*_**scenario_gen::select_scenario** property is used to determine which scenario descriptor, out of the available set of descriptors, is randomized next. The individual instances of the output stream are then created, by calling the *class-name*_**scenario::apply()** method of the randomized scenario descriptor.

By default, this property contains one instance of the atomic scenario descriptor *class-name*_**atomic_scenario**. Out of the box, the scenario generator generates individual random descriptors.

The **vmm_data::stream_id** property of the randomized instance is assigned the value of the stream identifier of the generator, before randomization. The **vmm_data::scenario_id** property of the randomized instance is assigned a unique value, before randomization. It will be reset to 0, when the generator is reset, and after the specified number of instances or scenarios are generated.

## Example

*Example B-42*

```
program test_scenario;
   ...
   atm_cell_scenario_gen atm_gen =
      new("Atm Scenario Gen", 12);
   my_scenario test_scen = new();
   ...
   initial
   begin
      ...
      atm_gen.scenario_set.delete();
      atm_gen.scenario_set.push_back(test_scen);
      atm_gen.stop_after_n_scenarios = 10;
      atm_gen.start_xactor();
      ...
   end
   ...
endprogram
```

## vmm_scenario_gen::select_scenario

Determines which scenario descriptor will be randomized next.

### SystemVerilog

```
vmm_scenario_election#(T,text) select_scenario;
```

### OpenVera

Not supported.

### Description

References the scenario descriptor selector that is repeatedly randomized to determine which scenario descriptor, out of the available set of scenario descriptors, will be randomized next.

By default, a round-robin selection process is used. The constraint blocks or randomized properties in this instance can be turned-off, or the instance can be replaced with a user-defined extension, to modify the election rules.

### Example

*Example B-43*

```
program test_scenario;
   ...
   atm_cell_scenario_gen atm_gen =
      new("Atm Scenario Gen", 12);
   my_scenario scen;
   ...
   initial
   begin
      atm_gen.scenario_set.push_back(scen);
```

```
      atm_gen.stop_after_n_scenarios = 10;
      atm_gen.start_xactor();
      ...
      if(atm_gen.select_scenario == null)
         `vmm_note(log,"Failed to create select_scenario
         instance for ATM Scenario Generator.");
   end
   ...
endprogram
```

## vmm_scenario_gen::stop_after_n_insts

Stops generation, after the specified number of transaction or data descriptor instances are generated.

### SystemVerilog

```
int unsigned stop_after_n_insts;
```

### OpenVera

Not supported.

### Description

The generator stops after the specified number of transaction or data descriptor instances are generated, and consumed by the output channel. The generator must be reset, before it can be restarted. If the value of this property is 0, the generator does not stop on its own, based on the number of generated instances (but may still stop, based on the number of generated scenarios).

The default value of this property is 0.

### Example

*Example B-44*

```
program test_scenario;
   ...
   atm_cell_scenario_gen atm_gen =
      new("Atm Scenario Gen", 12);
   ...
   initial
   begin
      atm_gen.stop_after_n_insts = 10;
```

```
      atm_gen.start_xactor();
      ...
   end
   ...
endprogram
```

## vmm_scenario_gen::stop_after_n_scenarios

Stops generation, after the specified number of scenarios are generated.

### SystemVerilog

```
int unsigned stop_after_n_scenarios;
```

### OpenVera

Not supported.

### Description

The generator stops after the specified number of scenarios are generated, and entirely consumed by the output channel. The generator must be reset, before it can be restarted. If the value of this property is 0, the generator does not stop on its own, based on the number of generated scenarios (but may still stop, based on the number of generated instances).

The default value of this property is 0.

### Example

*Example B-45*

```
program test_scenario;
   ...
   atm_cell_scenario_gen atm_gen =
      new("Atm Scenario Gen", 12);
   ...
   initial
   begin
      atm_gen.stop_after_n_scenarios = 10;
```

```
      atm_gen.start_xactor();
      ...
   end
   ...
endprogram
```

## vmm_scenario_gen::unregister_scenario()

Unregisters a scenario descriptor.

### SystemVerilog

```
virtual function bit unregister_scenario(
     vmm_ss_scenario_base scenario);
```

### OpenVera

Not supported.

### Description

Completely unregisters the specified scenario descriptor and returns `TRUE`, if it exists in the registry. The unregistered scenario is also removed from the scenario set.

### Example

*Example B-46*

```
`vmm_scenario_gen(atm_cell, "atm trans")

program test_scenario;
   atm_cell_scenario_gen atm_gen =
      new("Atm Scenario Gen", 12);
   atm_cell_scenario atm_scenario = new;
    ...
   initial begin
      if(atm_gen.unregister_scenario(atm_scenario))
         vmm_log(log,"Scenario has been unregistered \n");
      else
         vmm_log(log,"Unable to unregister scenario\n");
   end
endprogram
```

## vmm_scenario_gen::unregister_scenario_by_name()

Unregisters a scenario descriptor.

### SystemVerilog

```
virtual function vmm_scenario
unregister_scenario_by_name(string name)
```

### OpenVera

Not supported.

### Description

Unregisters the scenario under the specified name, and returns the unregistered scenario descriptor. Returns `NULL`, if there is no scenario registered under the specified name.

The unregistered scenario descriptor is removed from the scenario set, if it is not also registered under another name.

### Example

*Example B-47*

```
`vmm_scenario_gen(atm_cell, "atm trans")

program test_scenario;
   atm_cell_scenario_gen atm_gen =
      new("Atm Scenario Gen", 12);
   atm_cell_scenario atm_scenario = new;
   atm_cell_scenario buffer_scenario = new;
   ...
   initial begin
      ...
      buffer_scenario =
```

```
atm_gen.unregister_scenario_by_name("PARENT SCEN");
      if(buffer_scenario != null)
         vmm_log(log,"Scenario has been unregistered \n");
         ...
      else
         vmm_log(log,"Returned null value\n");
      ...
   end
endprogram
```

## ‘vmm_scenario_gen

Macro to define a scenario generator class to generate sequences of related instances.

### SystemVerilog

```
‘vmm_scenario_gen(class_name, "Class Description")
```

### OpenVera

Not supported.

### Description

Defines a scenario generator class to generate sequences of related instances of the specified class. The specified class must be derived from the **`vmm_data`** class, and the *`class-name`*`_`**`channel`** class must exist. It must also contain a constructor with no arguments, or that contain default values for all of its arguments.

The macro defines classes named

- *class-name*`_`**`scenario_gen`**
- class-name`_`**`scenario`**
- class-name_**scenario_election**

*class-name*`_`**`scenario_gen_callbacks`**

### Example

*Example B-48*

```
class atm_cell extends vmm_data;
```

```
   ...
endclass

`vmm_scenario_gen(atm_cell, "atm trans")
```

## ‘vmm_scenario_gen_using()

Defines a scenario generator class to generate sequences of related instances.

### SystemVerilog

```
‘vmm_scenario_gen_using( class-name , channel-type,
    "Class Description")
```

### OpenVera

Not supported.

### Description

Defines a scenario generator class to generate sequences of related instances of the specified class, using the specified *class-name*_**channel** output channel. The generated class must be compatible with the specified channel type, and both must exist.

This macro should be used only when generating instances of a derived class that must be applied to a channel of the base class.

### Example

*Example B-49*

```
class atm_cell extends vmm_data;
   ...
endclass
// `vmm_scenario_gen(atm_cell, "atm trans")
// You cannot use both `vmm_scenario_gen and
// `vmm_scenario_gen_using.
`vmm_scenario_gen_using(atm_cell,atm_cell_channel,
   "atm_cell")
```

# <*class-name*>_scenario

This class implements a base class for describing scenarios or sequences of transaction descriptors. This class named *class-name*_**scenario** is automatically declared and implemented for any user-specified class named *class-name* by the scenario generator macro, using a process similar to the `**vmm_channel** macro.

## Summary


- <class-name>_scenario::allocate_scenario() ..... page B-140
- <class-name>_scenario::apply() ..... page B-142
- <class-name>_scenario::define_scenario() ..... page B-143
- <class-name>_scenario::fill_scenario() ..... page B-144
- <class-name>_scenario::items[] ..... page B-145
- <class-name>_scenario::length ..... page B-147
- <class-name>_scenario::log ..... page B-148
- <class-name>_scenario::redefine_scenario() ..... page B-149
- <class-name>_scenario::repeat_thresh ..... page B-151
- <class-name>_scenario::repeated ..... page B-152
- <class-name>_scenario::scenario_id ..... page B-153
- <class-name>_scenario::scenario-kind ..... page B-154
- <class-name>_scenario::scenario_name() ..... page B-155
- <class-name>_scenario::stream_id ..... page B-156
- <class-name>_scenario::using ..... page B-157

## <class-name>_scenario::allocate_scenario()

Allocates a new set of instances in the **`items`** property.

### SystemVerilog

```
function void
   allocate_scenario(class-name using = null);
```

### OpenVera

Not supported.

### Description

Allocates a new set of instances in the **`items`** property, up to the maximum number of items that are in the maximum-length scenario. Any instance previously located in the **`items`** array is replaced. If a reference to an instance is specified in the **`using`** argument, the array is filled by calling the **`vmm_data::copy()`** method on the specified instance. Otherwise, the array is filled with new instance of the *`class-name`* class.

### Example

*Example B-50*

```
class my_scenario extends atm_cell_scenario;
   ...
   rand write_scenario scen1;
   ...
   constraint test {
      if (scenario_kind == ATM) {
         repeated == 4;
         foreach(items[i]) {
            ...
```

```
            items[i].kind == atm_cell::WRITE;
            items[i].addr == 64'hfff;
            ...
         }
      }
   }
   ...
   virtual task apply(atm_cell_channel chan,
         ref int unsigned n_insts);
      super.apply(chan,n_insts);
      this.allocate_scenario(tr);
      scen1.apply(chan, n_insts);
      ...
   endtask
   ...
endclass
```

## <class-name>_scenario::apply()

Applies the items in the scenario descriptor to an output channel.

### SystemVerilog

```
virtual task apply(class-name_channel channel,
    ref int unsigned n-insts);
```

### OpenVera

Not supported.

### Description

Applies the items in the scenario descriptor to the specified output channel, and returns when they are consumed by the channel. The *n-insts* argument is set to the number of instances that were consumed by the channel. By default, copies the values of the *items* array using the **vmm_data::copy()** method.

This method may be overloaded to define procedural scenarios.

### Example

*Example B-51*

```
class dut_ms_sequence;
   rand eth_frame_sequence to_phy;
   rand eth_frame_sequence to_mac;
   rand wb_cycle_sequence  to_host;
   virtual task apply(eth_frame_channel to_phy_chan,
      eth_frame_channel to_mac_chan,
      wb_cycle_channel wb_chan);
   endtask
endclass: dut_ms_sequence
```

## <class-name>_scenario::define_scenario()

Defines a new scenario.

### SystemVerilog

```
function int unsigned
                define_scenario(string name,
    int unsigned max-len = 0);
```

### OpenVera

Not supported.

### Description

Defines a new scenario with the specified name, and the specified maximum number of transactions or data descriptors. Returns a unique scenario identifier that should be assigned to an **`int unsigned`** property.

### Example

*Example B-52*

```
class my_scenario extends atm_cell_scenario;
   ...
   function new();
      ...
      this.ATM = define_scenario("ATM read write", 6);
      ...
   endfunction
   ...
endclass
```

## <class-name>_scenario::fill_scenario()

Allocates new instances in the **`items`** property.

### SystemVerilog

```
function void fill_scenario(class-name using = null);
```

### OpenVera

Not supported.

### Description

Allocates new instances in the **`items`** property, up to the maximum number of items in the maximum-length scenario, in any *`null`* element of the array. Any instance, which is previously located in the **`items`** array is left untouched. If a reference to an instance is specified in the **`using`** argument, the array is filled by calling the **`vmm_data::copy()`** method on the specified instance. Otherwise, the array is filled with a new instance of the *`class-name`* class.

### Example

*Example B-53*

```
class my_scenario extends atm_cell_scenario;
   ...
   rand write_scenario scen1;
   ...
   virtual task apply(atm_cell_channel chan,
         ref int unsigned n_insts);
      this.fill_scenario(tr);
      scen1.apply(chan, n_insts);
   endtask
endclass
```

## <class-name>_scenario::items[]

Instances that are randomized to form the scenarios.

### SystemVerilog

```
rand class-name items[];
```

### OpenVera

Not supported.

### Description

Instances of user-specified *class-name* that are randomized to form the scenarios. Only elements from index 0 to *class-name*_**scenario***::length*-1 are part of the scenario.

The constraint blocks and **rand** attributes of the instances in the randomized array may be turned **ON** or **OFF** to modify the constraints on scenario items. They can also be replaced with extensions.

By default, the output stream is formed by copying the values of the items in this array, onto the output channel.

### Example

*Example B-54*

```
class my_scenario extends atm_cell_scenario;
   ...
   constraint test {
      if (scenario_kind == ATM) {
         length == 4;
         foreach(items[i]) {
            ...
```

```
            items[i].kind == atm_cell::WRITE;
            items[i].addr == 64'hfff;
            ...
        }
    }
  }
  ...
endclass
```

## <class-name>_scenario::length

Defines the randomized number of items in the scenario.

### SystemVerilog

```
rand int unsigned length;
```

### OpenVera

Not supported.

### Description

Defines how many instances in the *class-name*_**scenario::items[]** property are part of the scenario.

### Example

*Example B-55*

```
class my_scenario extends atm_cell_scenario;
   ...
   constraint test {
      if (scenario_kind == ATM) {
         ...
         length == 4;
         ...
      }
   }
   `vmm_note(log,$psprintf("Scenario Length %0d.",length));
   ...
endclass
```

## <class-name>_scenario::log

Message service interface to be used to issue generic messages.

### SystemVerilog

```
static vmm_log log = new(“class-name”,“class”);
```

### OpenVera

Not supported.

### Description

Message service interface to be used to issue generic messages, when the message service interface of the scenario generator is not available or in scope.

### Example

*Example B-56*

```
class atm_cell extends vmm_data;
   ...
endclass

`vmm_scenario_gen(atm_cell, "atm trans")

class my_scenario extends atm_cell_scenario;
   ...
   function new();
      `vmm_note(log,
        "Display is coming from atm_cell_scenario class.");
      ...
   endfunction
endclass
```

## <class-name>_scenario::redefine_scenario()

Redefines the name and maximum number of descriptors in a scenario.

### SystemVerilog

```
function void
   redefine_scenario(int unsigned scenario-kind,
     string name,
     int unsigned max-len);
```

### OpenVera

Not supported.

### Description

Redefines the name and maximum number of descriptors in a previously defined scenario. Used to redefine an existing scenario instead of creating a new one, and constrain the original scenario out of existence.

### Example

*Example B-57*

```
class my_scenario extends atm_cell_scenario;
   ...
   function new();
      ...
      this.ATM = define_scenario("ATM read write", 6);
      ...
   endfunction
   ...
  redefine_scenario(scenario_kind,"Redefined our scenario",
      10);
```

```
    ...
    `vmm_note(log,$psprintf({"After Redefining the
      scenario=>\n Scenario Name:","
      %0s and Max scenarios:
      %0d"},scenario_name(scenario_kind),
      get_max_length()));
    ...
endclass
```

## <class-name>_scenario::repeat_thresh

Threshold for the number of times to repeat a scenario.

### SystemVerilog

```
static int unsigned repeat_thresh;
```

### OpenVera

Not supported.

### Description

To avoid accidentally repeating a scenario many times, because the **`repeated`** property was left unconstrained. A warning message is generated, if the value of the **`repeated`** property is greater than the value specified in this property. The default value is 100.

### Example

*Example B-58*

```
class my_scenario extends atm_cell_scenario;
   function new();
      ...
      this.ATM = define_scenario("ATM read write", 6);
      repeat_thresh = 2;
   endfunction
   constraint test {
      repeated == 5;
   }
  // Here repeated > repeat_thresh so warning will be issued.
  // Warning: A scenario will be repeated 5 times...
   `vmm_note(log,$psprintf(
      "repeat_thresh scenarios: %0d.",repeat_thresh));
endclass
```

## <class-name>_scenario::repeated

Returns the number of times the items in the scenario are repeated.

### SystemVerilog

```
rand int unsigned repeated;
```

### OpenVera

Not supported.

### Description

A value of 0 indicates that the scenario is not repeated, hence is applied only once. The repeated instances in the scenario count toward the total number of instances generated, but only one scenario is considered generated, regardless of the number of times it is repeated.

### Example

*Example B-59*

```
class my_scenario extends atm_cell_scenario;
   ...
   constraint test {
      if (scenario_kind == ATM) {
         repeated == 4;
      }
   }
   `vmm_note(log,$psprintf(
      "Repeated Scenarios %0d.",repeated));
   ...
endclass
```

## <class-name>_scenario::scenario_id

Identifies the scenario.

### SystemVerilog

```
int scenario_id;
```

### OpenVera

Not supported.

### Description

Identifies the scenario within the stream. It is set by the scenario generator before the scenario descriptor is randomized, and incremented after each randomization. Can be used to express scenario-specific constraints. The scenario identifier is reset to 0 when the scenario generator is reset, or when the specified number of scenarios are generated.

### Example

*Example B-60*

```
class my_scenario extends atm_cell_scenario;
   ...
   `vmm_note(log,$psprintf("Scenario ID for
      atm_cell_scenario #%0d.",scenario_id));
   ...
endclass
```

## <class-name>_scenario::scenario-kind

Selects the identifier of the scenario that is generated.

### SystemVerilog

```
rand int unsigned scenario-kind;
```

### OpenVera

Not supported.

### Description

When randomized, selects the identifier of the scenario that is generated. Constrained to the known scenario identifiers defined, using the *class-name*_**scenario::define_scenario()** method. Can be constrained to modify the distribution of generated scenarios.

### Example

*Example B-61*

```
class my_scenario extends atm_cell_scenario;
   ...
   function new();
      this.ATM = define_scenario("ATM read write", 6);
      scenario_kind = this.ATM;
      ...
   endfunction
   ...
   `vmm_note(log,$psprintf(
      "Scenario Kind: %0d.",scenario_kind));
   ...
endclass
```

## <class-name>_scenario::scenario_name()

Returns the name associated with the specified scenario identifier.

### SystemVerilog

```
function string
            scenario_name(int unsigned scenario-kind);
```

### OpenVera

Not supported.

### Example

*Example B-62*

```
class my_scenario extends atm_cell_scenario;
   ...
   function new();
      ...
      this.ATM = define_scenario("ATM read write", 6);
      scenario_kind = this.ATM;
      ...
   endfunction
   ...
   `vmm_note(log,$psprintf("Scenario Name:
      %0s",scenario_name(scenario_kind)));
   ...
endclass
```

## <class-name>_scenario::stream_id

Identifies the stream.

### SystemVerilog

```
int stream_id;
```

### OpenVera

Not supported.

### Description

Identifies the stream. It is set by the scenario generator, before the scenario descriptor is randomized. Can be used to express stream-specific constraints.

### Example

*Example B-63*

```
class my_scenario extends atm_cell_scenario;
   ...
   function new();
      ...
      this.ATM = define_scenario("ATM read write", 6);
      ...
   endfunction
   ...
   `vmm_note(log,$psprintf(
      "Stream ID for atm_cell_scenario #%0d.",stream_id));
   ...
endclass
```

## <class-name>_scenario::using

Instance used in **`pre_randomize()`** when invoking the **`fill_scenario()`** method.

### SystemVerilog

*`class-name`* `using;`

### OpenVera

Not supported.

### Description

Instance used in the default implementation of the **`pre_randomize()`** method, when invoking the **`fill_scenario()`** method. Sets to *`null,`* by default. Can be replaced by an instance of a derived class, to subject the items of the scenario to different constraints or content.

### Example

*Example B-64*

```
class my_scenario extends atm_cell_scenario;
   function new(atm_cell tr);
      ...
      this.ATM = define_scenario("ATM read write", 6);
      this.using = tr;
      ...
   endfunction
endclass
my_scenario atm;
// It will call the fill_scenario method with using object.
atm.pre_randomize();
```

# <*class-name*>_atomic_scenario

This class implements a predefined atomic scenario descriptor. An atomic scenario is composed of a single, unconstrained transaction or data descriptor. The *class-name*`_atomic_scenario` class is automatically implemented for any user-specified class, *class-name,* by the scenario generator macro, using a process similar to the **`'vmm_channel`** macro.

## Summary


- <class-name>_atomic_scenario::ATOMIC ..... page B-159
- <class-name>_atomic_scenario::atomic-scenario .... page B-160

## <class-name>_atomic_scenario::ATOMIC

Identifier for the atomic scenario.

### SystemVerilog

```
int unsigned ATOMIC;
```

### OpenVera

Not supported.

### Description

Symbolic scenario identifier for the atomic scenario, described by this descriptor. The atomic scenario is a single, random, unconstrained, and transaction descriptor (that is, an atomic descriptor).

### Example

*Example B-65*

```
class my_scenario extends atm_cell_atomic_scenario;
   ...
   constraint repetition {
      if (scenario_kind == ATOMIC) {
         length == 2;
         repeated <  122;
      }
   }
   function new();
      ...
      redefine_scenario(this.ATOMIC, "my_scenario", 2);
      ...
   endfunction
endclass
```

## <class-name>_atomic_scenario::atomic-scenario

Constraints of the atomic scenario.

### SystemVerilog

```
constraint atomic_scenario;
```

### OpenVera

Not supported.

### Description

Specifies the constraints of the atomic scenario. By default, the atomic scenario is a single, unrepeated, and unconstrained item. This constraint block may be overridden to redefine the atomic scenario.

### Example

*Example B-66*

```
class my_scenario extends atm_cell_atomic_scenario;
   constraint atomic_scenario {
      if (scenario_kind == ATOMIC) {
         length == 2;
         repeated <  122;
      }
   }
  // If you do not overwrite atomic_scenario constraint then
   // Scenario Length = 1
   // Repeated Scenario = 0
   `vmm_note(log,$psprintf("
       Scenario Length: %0d & Repeated Scenario: %0d",
       length,repeated);
endclass
```

# <*class-name*>_scenario_election

This class implements a random selection process for selecting the next scenario descriptor, from a set of available descriptors, to be randomized next. The *class-name*_**scenario_election** class is automatically implemented for any user-specified class, *class-name,* by the scenario generator macros, using a process similar to the **`vmm_channel** macro.

## Summary


- <class-name>_scenario_election::last_selected[$] . page B-162
- <class-name>_scenario_election::n_scenarios ..... page B-163
- <class-name>_scenario_election::next_in_set ..... page B-164
- <class-name>_scenario_election::round_robin ..... page B-165
- <class-name>_scenario_election::scenario_id ..... page B-166
- <class-name>_scenario_election::scenario_set[$] .. page B-167
- <class-name>_scenario_election::select ........... page B-168
- <class-name>_scenario_election::stream_id ........ page B-169

## <class-name>_scenario_election::last_selected[$]

Returns the history of the last scenario selections.

### SystemVerilog

```
int unsigned last_selected[$];
```

### OpenVera

Not supported.

### Description

Returns the history (maximum of 10) of last scenario selections. Can be used to express constraints based on the historical distribution of the selected scenarios (for example, “Never select the same scenario twice in a row.”).

### Example

*Example B-67*

```
class scen_election extends atm_cell_scenario_election;
   ...
endclass

program test_scenario;
   scen_election elect;
   ...
   initial
   begin
      elect.last_selected =
         gen.select_scenario.last_selected;
   end
   ...
endprogram
```

## <class-name>_scenario_election::n_scenarios

Number of available scenario descriptors in the scenario set.

### SystemVerilog

```
int unsigned n_scenarios;
```

### OpenVera

Not supported.

### Description

The final value of the *select* property must be in the [0:*n_scenarios*-1] range.

### Example

*Example B-68*

```
class scen_election extends atm_cell_scenario_election;
  ...
endclass

program test_scenario;
   scen_election a_scen;
   initial
   begin
      a_scen.n_scenarios = 5;
      ...
   end
   ...
endprogram
```

## <class-name>_scenario_election::next_in_set

The next scenario in a round-robin selection process.

### SystemVerilog

```
int unsigned next_in_set;
```

### OpenVera

Not supported.

### Description

The next scenario descriptor index that would be selected in a round-robin selection process. Used by the **`round_robin`** constraint block.

### Example

*Example B-69*

```
class scen_election extends atm_cell_scenario_election;
   ...
   constraint round_robin {
      select == next_in_set;
   }
   ...
endclass
```

## <class-name>_scenario_election::round_robin

Constrains the scenario selection process to a round-robin selection.

### SystemVerilog

```
constraint round_robin;
```

### OpenVera

Not supported.

### Description

This constraint block may be turned-off to produce a random scenario selection process, or allow a different constraint block to define a different scenario selection process.

### Example

*Example B-70*

```
class scen_election extends atm_cell_scenario_election;
   ...
   constraint round_robin {
      select == next_in_set;
   }
   ...
endclass
```

## <class-name>_scenario_election::scenario_id

Identifies the scenario within the stream.

### SystemVerilog

```
int scenario_id;
```

### OpenVera

Not supported.

### Description

It is set by the scenario generator before the scenario selector is randomized, and incremented after each randomization. Can be used to express scenario-specific constraints. The scenario identifier is reset to 0 when the scenario generator is reset, or when the specified number of scenarios are generated.

### Example

*Example B-71*

```
`vmm_scenario_gen(atm_cell, "ATM Cell")
class scen_election extends atm_cell_scenario_election;
   constraint con_select {
      if (this.scenario_id % 5 == 0)
         begin
            select dist {
               0 := 3,
               1 := 1
            };
         end
   }
endclass
```

## <class-name>_scenario_election::scenario_set[$]

The set of scenario descriptors.

### SystemVerilog

```
class-name_scenario scenario_set[$];
```

### OpenVera

Not supported.

### Description

The available set of scenario descriptors. Can be used to procedurally determine, which scenario to select or to express constraints based on the scenario descriptors.

### Example

*Example B-72*

```
class scen_election extends atm_cell_scenario_election;
   ...
endclass

program test_scenario;
   ...
   initial
   begin
      scen_election elect;
      atm_cell_scenario_gen gen = new("Scenario Gen");
      gen.select_scenario.scenario_set =
        elect.scenario_set;
      ...
   end
endprogram
```

## <class-name>_scenario_election::select

The index of the selected scenario to be randomized next.

### SystemVerilog

```
rand int select;
```

### OpenVera

Not supported.

### Description

The index, within the **`scenario_set`** array, of the selected scenario descriptor to be randomized next.

### Example

*Example B-73*

```
class scen_election extends atm_cell_scenario_election;
   ...
   constraint distribution{
      select dist {0 := 3,
                   1 := 1
      };
   }
   ...
endclass
```

## <class-name>_scenario_election::stream_id

Stream identifier.

### SystemVerilog

```
int stream_id;
```

### OpenVera

Not supported.

### Description

It is set by the scenario generator to the value of the generator stream identifier, before the scenario selector is randomized. Can be used to express stream-specific constraints.

### Example

*Example B-74*

```
`vmm_scenario_gen(atm_cell, "ATM Cell")

class scen_election extends atm_cell_scenario_election;
   ...
endclass

program test_scenario;
   scen_election elect;
   ...
   initial
   begin
      elect.stream_id =0;
      ...
   end
endprogram
```

# <*class-name*>_scenario_gen_callbacks

This class implements a façade for callback containments for the scenario generator transactor. The *class-name*_**scenario_gen_callbacks** class is automatically implemented for any user-specified class, *class-name,* by the scenario generator macro, using a process similar to the **`vmm_channel** macro.

## Summary


- <class-name>_scenario_gen_callbacks::post_scenario_gen() page B-171
- <class-name>_scenario_gen_callbacks::pre_scenario_randomize() page B-173

## <class-name>_scenario_gen_callbacks::post_scenario_gen()

Callback invoked by the generator, after a scenario is randomized.

### SystemVerilog

```
virtual task post_scenario_gen(
     class-name_scenario_gen gen,
     class-name_scenario scenario,
     ref bit dropped);
```

### OpenVera

Not supported.

### Description

Callback method invoked by the generator after a new scenario is randomized, but before it is applied to the output channel. The *gen* argument refers to the generator instance that is invoking the callback method. The *scenario* argument refers to the newly randomized scenario that can be modified. Note that any modifications of the randomization state of the scenario descriptor, such as turning constraint blocks ON or OFF, remains in effect the next time the scenario descriptor is selected to be randomized. If the value of the *dropped* argument is set to non-zero, then the generated instance is not applied to the output channel.

### Example

*Example B-75*

```
`vmm_scenario_gen(atm_cell, "ATM Cell")
class atm_scen_callbacks extends
atm_cell_scenario_gen_callbacks;
```

```
  virtual task post_scenario_gen(atm_cell_scenario_gen gen,
         atm_cell_scenario scenario,
         ref bit dropped);
       ...
   endtask
   ...
endclass
```

## <class-name>_scenario_gen_callbacks::pre_scenario_randomize()

Callback invoked by the generator after a scenario is selected.

### SystemVerilog

```
virtual task pre_scenario_randomize(
     class-name_scenario_gen gen,
     ref class-name_scenario scenario);
```

### OpenVera

Not supported.

### Description

Callback method invoked by the generator after a new scenario is selected, but before it is randomized. The *gen* argument refers to the generator instance that is invoking the callback method. The *scenario* argument refers to the newly selected scenario descriptor, which can be modified. Note that any modifications of the randomization state of the scenario descriptor, such as turning constraint blocks ON or OFF, remains in effect the next time the scenario descriptor is selected to be randomized. If the reference to the scenario descriptor is set to *null*, then the scenario will not be randomized and a new scenario will be selected.

To minimize memory allocation and collection, it is possible that the elements of the scenarios may not be allocated. Use the *class-name*_**scenario::allocate_scenario()** or the *class-name*_**scenario::fill_scenario()** to allocate the elements of the scenario, if necessary.

## Example

*Example B-76*

```
`vmm_scenario_gen(atm_cell, "ATM Cell")
   class atm_scen_callbacks extends
   atm_cell_scenario_gen_callbacks;

   virtual task pre_scenario_randomize(
         atm_cell_scenario_gen gen,
         ref atm_cell_scenario scenario);
      ...
   endtask
   ...
endclass
```

# vmm_scheduler

Channels are point-to-point transaction descriptor transfer mechanisms. If multiple sources are adding descriptors to a single channel, then the descriptors are interleaved with the descriptors from the other sources, in a fair but uncontrollable way. If a multi-point-to-point mechanism is required to follow a specific scheduling algorithm, a **vmm_scheduler** component can be used to identify which source stream should next be forwarded to the output stream.

This class is based on the **vmm_xactor** class.

## Summary


- vmm_scheduler::log ..... page B-178
- vmm_scheduler::new() ..... page B-179
- vmm_scheduler::new_source() ..... page B-180
- vmm_scheduler::out_chan ..... page B-181
- vmm_scheduler::randomized_sched ..... page B-182
- vmm_scheduler::reset_xactor() ..... page B-183
- vmm_scheduler::sched_off() ..... page B-184
- vmm_scheduler::sched_on ..... page B-185
- vmm_scheduler::schedule() ..... page B-186
- vmm_scheduler::set_output() ..... page B-188
- vmm_scheduler::start_xactor() ..... page B-189
- vmm_scheduler::stop_xactor() ..... page B-190

## vmm_scheduler::get_object()

Extracts the next scheduled transaction descriptor.

### SystemVerilog

```
virtual protected task get_object(
                       output vmm_data obj,
                       input vmm_channel source,
                       input int unsigned input_id,
                       input int offset);
```

### OpenVera

Not supported.

### Description

This method is invoked by the default implementation of the **`vmm_scheduler::schedule()`** method to extract the next scheduled transaction descriptor from the specified input channel, at the specified offset within the channel. Overloading this method allows access to or replacement of the descriptor that is about to be scheduled. User-defined extensions can be used to introduce errors by modifying the object, interfere with the scheduling algorithm by substituting a different object, or recording of the schedule into a functional coverage model.

Any object that is returned by this method, through the *obj* argument, must be either internally created or physically removed from the input source using the **`vmm_channel::get()`** method. If a reference to the object remains in the input channel (for example, by using the **`vmm_channel::peek()`** or

**`vmm_channel::activate()`** method), then it is liable to be scheduled more than once, as the mere presence of an instance in any of the input channel makes it available to the scheduler.

## Example

*Example B-77*

```
vmm_data  data_obj;
int unsigned input_ids[$];
...
task start();
   ...
   #1;
   scheduler.start_xactor();
   input_ids = {0,1};
   scheduler.schedule(data_obj,sources,input_ids);
   scheduler.get_object(data_obj,chan_2,1,0);
   ...
endtask
```

## vmm_scheduler::log

Message service interface for this scheduler.

### SystemVerilog

```
vmm_log log;
```

### OpenVera

Not supported.

### Description

Sets by the constructor, and uses the name and instance name specified in the constructor.

### Example

*Example B-78*

```
class atm_scheduler extends vmm_scheduler ;
   vmm_log log;
   function new(string name, string instance,
         vmm_channel out_chan, int instance_id = -1);
      super.new(name,instance,out_chan,instance_id);
      log = new (name, instance);
      ...
   endfunction
   ...
endclass
```

## vmm_scheduler::new()

Creates an instance of a channel scheduler.

### SystemVerilog

```
function new(string name,
     string instance,
     vmm_channel destination,
     int instance_id = -1, vmm_object parent = null);
```

### OpenVera

Not supported.

### Description

Creates a new instance of a channel scheduler object with the specified name, instance name, destination channel, and optional instance identifier. The destination can be assigned to null and set later by using “vmm_scheduler::set_output()” .

### Example

*Example B-79*

```
class atm_subenv extends vmm_subenv;
   atm_scheduler scheduler;
   atm_cell_channel chan_2;
   ...
   task sub_build();
      chan_2  = new("chan_2", "gen");
      scheduler = new("schedular","subenv",chan_2,1);
      ...
   endtask
endclass
```

## vmm_scheduler::new_source()

Adds the channel instance to the scheduler.

### SystemVerilog

```
virtual function int new_source(vmm_channel chan);
```

### OpenVera

Not supported.

### Description

Adds the specified channel instance, as a new input channel to the scheduler. This method returns an identifier for the input channel that must be used to modify the configuration of the input channel or -1, if an error occurred.

Any user extension of this method must call the **`super.new_source()`** method.

### Example

*Example B-80*

```
int int_id;
atm_cell_channel sources[$];
function build();
   ...
   sources.push_back(chan_2);
   sources.push_back(chan_3);
   int_id = scheduler.new_source(chan_1);
   int_id = scheduler.new_source(chan_2);
   ...
endfunction
```

## vmm_scheduler::out_chan

Reference to the output channel.

### SystemVerilog

```
protected vmm_channel out_chan;
```

### OpenVera

Not supported.

### Description

Set by the constructor.

### Example

*Example B-81*

```
class atm_scheduler extends vmm_scheduler ;
   function new(string name, string instance,
      vmm_channel out_chan, int instance_id = -1);
      ...
      this.out_chan = out_chan;
      ...
   endfunction
   ...
endclass
```

## vmm_scheduler::randomized_sched

Factory instance randomized by the default implementation of the **`vmm_scheduler::schedule()`** method.

### SystemVerilog

```
vmm_scheduler_election randomized_sched;
```

### OpenVera

Not supported.

### Description

Can be replaced with user-defined extensions, to modify the election rules.

### Example

*Example B-82*

```
class atm_scheduler extends vmm_scheduler ;
   ...
   function new(string name, string instance,
      vmm_channel out_chan, int instance_id = -1);
      ...
    randomized_sched.id_history[instance_id] = instance_id;
      ...
   endfunction
   ...
endclass
```

## vmm_scheduler::reset_xactor()

Resets this **vmm_scheduler** instance.

### SystemVerilog

```
virtual function void
      reset_xactor(vmm_xactor::reset_e rst_typ = SOFT_RST);
```

### OpenVera

Not supported.

### Description

The output channel and all input channels are flushed. If a **HARD_RST** reset type is specified, then the scheduler election factory instance in the **randomized_sched** property is replaced with a new default instance.

### Example

*Example B-83*

```
class atm_env extends vmm_env;
   ...
   task reset_dut();
      scheduler.reset_xactor();
      ...
   endtask
   ...
endclass
```

## vmm_scheduler::sched_off()

Turns-off scheduling from the specified input channel.

### SystemVerilog

```
virtual function void sched_off(int unsigned input-id);
```

### OpenVera

Not supported.

### Description

By default, scheduling from an input channel is on. When scheduling is turned off, the input channel is not flushed and the scheduling of new transaction descriptors from that source channel is inhibited. The scheduling of descriptors from that source channel is resumed, as soon as scheduling is turned on.

Any user extension of this method should call the **`super.sched_off()`** method.

## vmm_scheduler::sched_on

Turns-on scheduling from the specified input channel.

### SystemVerilog

```
virtual function void sched_on(int unsigned input-id);
```

### OpenVera

Not supported.

### Description

By default, scheduling from an input channel is on. When scheduling is turned off, the input channel is not flushed and the scheduling of new transaction descriptors from that source channel is inhibited. The scheduling of descriptors from that source channel is resumed, as soon as scheduling is turned on.

Any user extension of this method should call the **`super.sched_on()`** method.

## vmm_scheduler::schedule()

Creates scheduling components with different rules.

### SystemVerilog

```
virtual protected task
                  schedule(output vmm_data obj,
                  input vmm_channel sources[$],
                  int unsigned input_ids[$]);
```

### OpenVera

Not supported.

### Description

Overloading this method allows the creation of scheduling components with different rules. It is invoked for each scheduling cycle. The transaction descriptor returned by this method in the *obj* argument is added to the output channel. If this method returns *null*, no descriptor is added for this scheduling cycle. The input channels provided in the *sources* argument are all the currently non-empty ON input channels. Their corresponding input identifier is found in the *input_ids* argument.

New scheduling cycles are attempted, whenever the output channel is not full. If no transaction descriptor is scheduled from any of the currently non-empty source channels, then the next scheduling cycle will be delayed until an additional ON source channel becomes non-empty. Lock-up occurs, if there are no empty input channels and no OFF channels.

The default implementation of this method randomizes the instance found in the **`randomized_sched`** property.

## Example

*Example B-84*

```
vmm_data  data_obj;
int unsigned input_ids[$];
...
task start();
   ...
   #1;
   scheduler.start_xactor();
   input_ids = {0,1};
   scheduler.schedule(data_obj,sources,input_ids);
   ...
endtask
...
```

## vmm_scheduler::set_output()

Specifies the channel as the destination if not set previously.

### System Verilog

```
function void set_output(vmm_channel destination);
```

### Open Vera

Not supported

### Description

Identifies the channel as the destination of the scheduler if the destination is not set previously. If destination is already set, then a warning is issued stating that this particular call has been ignored.

### Example

*Example B-85*

```
class atm_env extends vmm_group;
...
void function build_ph();
     scheduler = new("schedular","subenv",null,1);
                       ...
                    endfunction
...
void function connect_ph();
    scheduler.set_output(out_chan);
...
endfunction
  ...
endclass
```

## vmm_scheduler::start_xactor()

Starts this **`vmm_scheduler`** instance.

### SystemVerilog

```
virtual function void start_xactor();
```

### OpenVera

Not supported.

### Description

The scheduler can be stopped. Any extension of this method must call **`super.start_xactor()`**.

### Example

*Example B-86*

```
class atm_env extends vmm_env;
   ...
   task start();
      scheduler.start_xactor();
      ...
   endtask
   ...
endclass
```

## vmm_scheduler::stop_xactor()

Suspends this **`vmm_scheduler`** instance.

### SystemVerilog

```
virtual function void stop_xactor();
```

### OpenVera

Not supported.

### Description

The scheduler can be restarted. Any extension of this method must the call **`super.stop_xactor() method`**.

### Example

*Example B-87*

```
class atm_env extends vmm_env;
   ...
   task stop();
      scheduler.stop_xactor();
      ...
   endtask
   ...
endclass
```

# vmm_scheduler_election

This class implements a round-robin election process by default. In its current form, turning it into a random election process requires that this class be extended. To simplify this process, you need to just turn-off the **default_round_robin** constraint block.

The following class properties should be read or added:

- **"vmm_scheduler_election::next_idx"**
- **"vmm_scheduler_election::source_idx"**
- **"vmm_scheduler_election::obj_offset"**

## Summary


- vmm_scheduler_election::default_round_robin ..... page B-192
- vmm_scheduler_election::election_id ..... page B-193
- vmm_scheduler_election::id_history[$] ..... page B-194
- vmm_scheduler_election::ids[$] ..... page B-195
- vmm_scheduler_election::instance_id ..... page B-196
- vmm_scheduler_election::n_sources ..... page B-197
- vmm_scheduler_election::next_idx ..... page B-198
- vmm_scheduler_election::obj_history[$] ..... page B-199
- vmm_scheduler_election::obj_offset ..... page B-200
- vmm_scheduler_election::post_randomize() ..... page B-201
- vmm_scheduler_election::source_idx ..... page B-202
- vmm_scheduler_election::sources[$] ..... page B-203

## vmm_scheduler_election::default_round_robin

Constraints required by the default round-robin election process.

### SystemVerilog

```
constraint default_round_robin;
```

### OpenVera

Not supported.

### Example

*Example B-88*

```
class atm_scheduler_election extends
vmm_scheduler_election;
   ...
   constraint default_round_robin {
      source_idx == next_idx;
   }

   constraint vmm_scheduler_election_valid {
      obj_offset == 0;
      source_idx >= 0;
      source_idx < n_sources;
   }
   ...
endclass
```

## vmm_scheduler_election::election_id

Incremented by the **`vmm_scheduler`** instance.

### SystemVerilog

```
int unsigned election_id;
```

### OpenVera

Not supported.

### Description

Incremented by the **`vmm_scheduler`** instance that is randomizing this object instance before every election cycle. Can be used to specified election-specific constraints.

### Example

*Example B-89*

```
class atm_scheduler extends vmm_scheduler ;
   ...
   function void my_disp();
      `vmm_note(log,$psprintf("election_id method
        %0d ",randomized_sched.election_id));
   endfunction
   ...
endclass
```

## vmm_scheduler_election::id_history[$]

A queue of input identifiers.

### SystemVerilog

```
int unsigned id_history[$];
```

### OpenVera

Not supported.

### Description

A queue of the (up to) 10 last input identifiers that were elected.

### Example

*Example B-90*

```
class atm_scheduler extends vmm_scheduler ;
   ...
   function void my_disp();
      `vmm_note(log,$psprintf(
         "id_history.size method %0d ",
         randomized_sched.id_history.size));
      foreach(randomized_sched.id_history[i])
         `vmm_note(log,$psprintf("ids[%0d] = %0d ",i,
         randomized_sched.id_history[i]));
   endfunction
   ...
endclass
```

## vmm_scheduler_election::ids[$]

Input identifiers corresponding to the source channels.

### SystemVerilog

```
int unsigned ids[$];
```

### OpenVera

Not supported.

### Description

Unique input identifiers corresponding to the source channels, at the same index, in the *sources* array.

### Example

*Example B-91*

```
class atm_scheduler extends vmm_scheduler ;
   ...
   function void my_disp();
      `vmm_note(log,$psprintf(
         "ids.size method %0d ",
         randomized_sched.ids.size));
      foreach(randomized_sched.ids[i])
        `vmm_note(log,$psprintf(
         "ids[%0d] = %0d ",i,
         randomized_sched.ids[i]));
   endfunction
   ...
endclass
```

## vmm_scheduler_election::instance_id

Instance identifier of a **`vmm_scheduler`** class instance.

### SystemVerilog

```
int instance_id;
```

### OpenVera

Not supported.

### Description

Instance identifier of the **`vmm_scheduler`** class instance that is randomizing this object instance. Can be used to specify the instance-specific constraints.

### Example

*Example B-92*

```
class atm_scheduler extends vmm_scheduler ;
   ...
   function void my_disp();
      `vmm_note(log,$psprintf(
         "instance_id method %0d ",
         randomized_sched.instance_id));
   endfunction
   ...
endclass
```

## vmm_scheduler_election::n_sources

Number of sources.

### SystemVerilog

```
int unsigned n_sources;
```

### OpenVera

Not supported.

### Description

Similar to the **`vmm_scheduler_election::sources.size()` method**.

### Example

*Example B-93*

```
class atm_scheduler extends vmm_scheduler ;
   ...
   function void my_disp();
      `vmm_note(log,$psprintf(
         "n_sources method %0d ",
         randomized_sched.n_sources));
   endfunction
   ...
endclass
```

## vmm_scheduler_election::next_idx

Assign to **`source_idx`** for a round-robin process.

### SystemVerilog

```
int unsigned next_idx;
```

### OpenVera

Not supported.

### Description

This is the value to assign to **`source_idx,`** to implement a round-robin election process.

### Example

*Example B-94*

```
class atm_scheduler extends vmm_scheduler ;
   ...
   function void my_disp();
      `vmm_note(log,$psprintf(
         "next_idx = %0d ",
         randomized_sched.next_idx));
   endfunction
   ...
endclass
```

## vmm_scheduler_election::obj_history[$]

A list of transaction descriptors.

### SystemVerilog

```
vmm_data obj_history[$];
```

### OpenVera

Not supported.

### Description

A list of the (up to) 10 last transaction descriptors that were elected.

### Example

*Example B-95*

```
class atm_scheduler extends vmm_scheduler ;
   ...
   function void my_disp();
      `vmm_note(log,$psprintf(
         "obj_history.size method %0d ",
         randomized_sched.obj_history.size));
      foreach(randomized_sched.obj_history[i])
        `vmm_note(log,$psprintf(
            "obj_history[%0d] = %0d ",i,
            randomized_sched.obj_history[i]));
   endfunction
   ...
endclass
```

## vmm_scheduler_election::obj_offset

Offset of the elected transaction descriptor, within the elected source channel.

### SystemVerilog

```
rand int unsigned obj_offset;
```

### OpenVera

Not supported.

### Description

Offset, within the source channel indicated by the **`source_idx`** property of the elected transaction descriptor, within the elected source channel. This property is constrained to be 0 in the **`vmm_scheduler_election_valid`** constraint block, to preserve ordering of the input streams.

### Example

*Example B-96*

```
class atm_scheduler extends vmm_scheduler ;
   ...
   function void my_disp();
      `vmm_note(log,$psprintf(
         "obj_offset = %0d",randomized_sched.obj_offset));
   endfunction
   ...
endclass
```

## vmm_scheduler_election::post_randomize()

Performs the round-robin election.

### SystemVerilog

```
function void post_randomize();
```

### OpenVera

Not supported.

### Description

The default implementation of this method helps to perform the round-robin election.

### Example

*Example B-97*

```
class atm_scheduler_election extends
      vmm_scheduler_election;
   function void pre_randomize();
      default_round_robin.constraint_mode(0);
      vmm_scheduler_election_valid.constraint_mode(0);
      ...
   endfunction
endclass

class atm_scheduler extends vmm_scheduler ;
   atm_scheduler_election randomized_sched;
   ...
   function new(...)
       randomized_sched = new();
   endfunction
endclass
```

## vmm_scheduler_election::source_idx

Index in the **`sources`** array of the elected source channel.

### SystemVerilog

```
rand int unsigned source_idx;
```

### OpenVera

Not supported.

### Description

An index of –1 indicates no election. The **`vmm_scheduler_election_valid`** constraint block constrains this property to be in the 0 to **`sources.size()`**-1 range.

### Example

*Example B-98*

```
class atm_scheduler extends vmm_scheduler ;
   ...
   function void my_disp();
      `vmm_note(log,$psprintf(
         "source_idx = %0d",randomized_sched.source_idx));
   endfunction
   ...
endclass
```

## vmm_scheduler_election::sources[$]

Input source channels with transaction descriptors available to be scheduled.

### SystemVerilog

```
vmm_channel sources[$];
```

### OpenVera

Not supported.

### Example

*Example B-99*

```
class atm_scheduler extends vmm_scheduler ;
   ...
   function void my_disp();
      `vmm_note(log,$psprintf(
         "sources.size method %0d ",
         randomized_sched.sources.size));
   endfunction
   ...
endclass
```

# vmm_ss_scenario#(T)

Parameterized version of the VMM single stream scenario.

## SystemVerilog

```
class vmm_ss_scenario #(type T) extends
vmm_ss_scenario_base;
```

## Description

The parameterized single stream scenario is used by the parameterized scenario generator. It extends the `vmm_scenario`. You can extend this class to create a scenario.

## Example

```
class ahb_trans extends vmm_data;
   rand bit [31:0] addr;
   rand bit [31:0] data;
endclass

`vmm_channel(ahb_trans)
`vmm_scenario_gen(ahb_trans, "AHB Scenario Gen")

class user_scenario extends ahb_trans_scenario;
endclass
```

Is the same as:

```
class user_scenario extends vmm_ss_scenario#(ahb_trans);
endclass
```

# vmm_simulation

The `vmm_simulation` class extending from `vmm_unit` is a top-level singleton module that manages the end-to-end simulation timelines. It includes pre-test and post-test timelines with predefined pre-test and post-test phases. The predefined pre-test phases are `build`, `configure`, and `connect`. The predefined post-test phase is `final`.

## Example

```
program tb_top;
   class my_test extends vmm_test;
      ...
   endclass

   class my_env extends vmm_group;
      ...
   endclass

   initial begin
      my test test1 = new("test1");
      my_env env = new("env");
      vmm_simulation my_sim;
      my_sim = vmm_simulation :: get_sim();
      ...
   end
endprogram
```

## Summary


- vmm_simulation::allow_new_phases() ..... page B-206
- vmm_simulation::display_phases() ..... page B-207
- vmm_simulation::get_post_timeline() ..... page B-208
- vmm_simulation::get_pre_timeline() ..... page B-209
- vmm_simulation::get_sim() ..... page B-210
- vmm_simulation::get_top_timeline() ..... page B-211
- vmm_simulation::run_tests() ..... page B-212

## vmm_simulation::allow_new_phases()

Enables the addition of user-defined phases in timelines.

### SystemVerilog

```
static function void vmm_simulation::allow_new_phases(
   bit allow = 1)
```

### Description

Enables the addition of user-defined phases in timelines, if `allow` is true. If the insertion of a user-defined phase is attempted, when new phases are not allowed, an error message is issued.

By default, addition of user-defined phases are not allowed.

### Example

```
program tb_top;
   class my_test extends vmm_test;
      ...
   endclass
   class my_env extends vmm_group;
      ...
   endclass

   initial begin
      my test test1 = new("test1");
      my_env env = new("env");
      ...
      vmm_simulation::allow_new_phases();
      // insert new phases using
        // vmm_timeline::insert_phase();
   end
endprogram
```

## vmm_simulation::display_phases()

Displays how various phases in the various timelines will be executed.

### SystemVerilog

```
static function void vmm_simulation::display_phases()
```

### Description

Displays how various phases in the various timelines will be executed (that is, in sequence or in parallel). Should be invoked after the `build` phase.

### Example

```
program tb_top;
   class my_test extends vmm_test;
      virtual function void start_of_sim_ph();
         vmm_simulation::display_phases();
      endfunction
   endclass

   class my_env extends vmm_group;
   endclass

   initial begin
      my test test1 = new("test1");
      my_env env = new("env");
      ...
      vmm_simulation::run_tests();
   end
endprogram
```

## vmm_simulation::get_post_timeline()

Returns the post-test timeline.

### SystemVerilog

```
static function vmm_timeline
   vmm_simulation::get_post_timeline()
```

### Description

Returns the post-test timeline.

## vmm_simulation::get_pre_timeline()

Returns the pre-test timeline.

### SystemVerilog

```
static function vmm_timeline
   vmm_simulation::get_pre_timeline()
```

### Description

Returns the pre-test timeline.

## vmm_simulation::get_sim()

Returns the `vmm_simulation` singleton.

### SystemVerilog

```
static function vmm_simulation vmm_simulation::get_sim()
```

### Description

Returns the `vmm_simulation` singleton.

### Example

```
program tb_top;
   class my_test extends vmm_test;
      ...
   endclass

   class my_env extends vmm_group;
      ...
   endclass

   initial begin
      my test test1 = new("test1");
      my_env env = new("env");
      vmm_simulation my_sim;
      ...
      my_sim = vmm_simulation :: get_sim();
      ...
   end
endprogram
```

## vmm_simulation::get_top_timeline()

Returns the top-level test timeline.

### SystemVerilog

```
static function vmm_timeline
   vmm_simulation::get_top_timeline()
```

### Description

Returns the top-level test timeline.

### Example

```
program tb_top;
   class my_test extends vmm_test;
      ...
   endclass

   class my_env extends vmm_group;
      ...
   endclass

   initial begin
      my test test1 = new("test1");
      my_env env = new("env");
      vmm_timeline my_tl;
      ...
      my_tl = vmm_simulation::get_top_timeline();
      ...
   end
endprogram
```

## vmm_simulation::run_tests()

Run tests specified at runtime.

### SystemVerilog

```
task vmm_simulation::run_tests()
```

### Description

Run tests specified at runtime using the `+vmm_test` or `+vmm_test_file`, or runs default test

The following is the usage of `+vmm_test_file` and `+vmm_test` to specify testcase at runtime:

```
+vmm_test_file+<file name>    - will run list of tests
specified in the file (if concatenation is allowed, otherwise
issues a fatal message)
+vmm_test=<testname>+<testname>+...
Run list of specified tests
+vmm_test=<test name>
- run specific test
+vmm_test=ALL_TESTS - run all the registered tests (if
concatenation is allowed, otherwise issues a fatal message)
```

### Example

```
program tb_top;
   class my_test extends vmm_test;
   endclass
   class my_env extends vmm_group;
   endclass
   initial begin
      my test test1 = new("test1");
      my_env env = new("env");
....
      vmm_simulation::run_tests();
```

```
    end
endprogram
```

# vmm_subenv

This is a base class used to encapsulate a reusable sub-environment.

## Summary


- vmm_subenv::cleanup() ..... page B-215
- vmm_subenv::configured() ..... page B-216
- vmm_subenv::do_psdisplay() ..... page B-217
- vmm_subenv::do_start() ..... page B-218
- vmm_subenv::do_stop() ..... page B-219
- vmm_subenv::do_vote() ..... page B-220
- vmm_subenv::do_what_e ..... page B-221
- vmm_subenv::end_test ..... page B-222
- vmm_subenv::log ..... page B-223
- vmm_subenv::new() ..... page B-224
- vmm_subenv::report() ..... page B-226
- vmm_subenv::start() ..... page B-227
- vmm_subenv::stop() ..... page B-228
- `vmm_subenv_member_begin() ..... page B-229
- `vmm_subenv_member_channel*() ..... page B-230
- `vmm_subenv_member_end() ..... page B-232
- `vmm_subenv_member_enum*() ..... page B-233
- `vmm_subenv_member_scalar*() ..... page B-235
- `vmm_subenv_member_string*() ..... page B-237
- `vmm_subenv_member_subenv*() ..... page B-239
- `vmm_subenv_member_user_defined() ..... page B-241
- `vmm_subenv_member_vmm_data*() ..... page B-242
- `vmm_subenv_member_xactor*() ..... page B-244

## vmm_subenv::cleanup()

Verifies end-of-test conditions.

### SystemVerilog

```
virtual task cleanup();
```

### OpenVera

```
virtual task cleanup_t();
```

### Description

Stops the sub-environment (if not already stopped), and then verifies any end-of-test conditions.

The base implementation must be called using the `super.cleanup()`, by any extension of this method, in a user-defined extension of this base class.

### Example

*Example B-100*

```
class my_vmm_subenv extends vmm_subenv;
   ...
   virtual task cleanup()
      super.cleanup();
      ...
   endtask
   ...
endclass
```

## vmm_subenv::configured()

Indicates that the DUT is configured.

### SystemVerilog

```
protected function void configured();
```

### OpenVera

```
protected task configured();
```

### Description

Reports to the base class that the sub-environment and associated DUT are configured appropriately, and that the sub-environment is ready to be started.

This method must be called by a user-defined *`configured()`* method in the extension of this base class.

### Example

*Example B-101*

```
class my_vmm_subenv extends vmm_subenv;
   ...
   protected function void configured(...);
      // Configuration of sub environment and corresponding
     // portion of DUT
       ...
      super.configured();
   endfunction
   ...
endclass
```

## vmm_subenv::do_psdisplay()

Overrides the shorthand `psdisplay()` method.

### SystemVerilog

```
virtual function string do_psdisplay(string prefix = "")
```

### OpenVera

Not supported.

### Description

This method overrides the default implementation of the `vmm_subenv::psdisplay()` method, created by the `vmm_subenv` shorthand macros. If defined, it will be used instead of the default implementation.

### Example

*Example B-102*

```
class my_vmm_subenv extends vmm_subenv;
   ...
  `vmm_subenv_member_begin( my_vmm_subenv)
      ...
   `vmm_subenv_member_end( my_vmm_subenv)
  virtual function string do_psdisplay(string prefix = "");
      $sformat(do_psdisplay,"%s Printing sub environment
          members \n",prefix);
      ...
   endfunction
   ...
endclass
```

## vmm_subenv::do_start()

Overrides the shorthand `start()` method.

### SystemVerilog

```
protected virtual task do_start()
```

### OpenVera

Not supported.

### Description

This method overrides the default implementation of the `vmm_subenv::start()` method created by the `vmm_subenv` shorthand macros. If defined, it will be used instead of the default implementation.

### Example

*Example B-103*

```
class my_vmm_subenv extends vmm_subenv;
   ...
   `vmm_subenv_member_begin( my_vmm_subenv)
      ...
   `vmm_subenv_member_end( my_vmm_subenv)
   protected virtual task do_start();
      //vmm_subenv::start() operations
      ...
   endtask
   ...
endclass
```

## vmm_subenv::do_stop()

Overrides the shorthand `stop()` method.

### SystemVerilog

```
protected virtual task do_stop()
```

### OpenVera

Not supported.

### Description

This method overrides the default implementation of the `vmm_subenv::stop()` method created by the `vmm_subenv` shorthand macros. If defined, it will be used instead of the default implementation.

### Example

*Example B-104*

```
class my_vmm_subenv extends vmm_subenv;
   ...
   `vmm_subenv_member_begin( my_vmm_subenv)
      ...
   `vmm_subenv_member_end( my_vmm_subenv)
   protected virtual task do_stop();
      //vmm_subenv::stop() operations
      ...
   endtask
   ...
endclass
```

## vmm_subenv::do_vote()

Overrides the shorthand voter registration.

### SystemVerilog

```
protected virtual task do_vote()
```

### OpenVera

Not supported.

### Description

This method overrides the default implementation of the voter registration, created by the `vmm_subenv` shorthand macros. If defined, it will be used instead of the default implementation.

### Example

*Example B-105*

```
class my_vmm_subenv extends vmm_subenv;
   ...
   `vmm_subenv_member_begin( my_vmm_subenv)
      ...
   `vmm_subenv_member_end( my_vmm_subenv)
   protected virtual task do_vote();
      //Register with this.end_vote
      ...
   endtask
   ...
endclass
```

## vmm_subenv::do_what_e

Specifies which methods are to be provided by a shorthand implementation.

### SystemVerilog

```
enum {DO_PRINT, DO_START, DO_STOP,
      DO_VOTE, DO_ALL} do_what_e;
```

### OpenVera

Not supported.

### Description

Used to specify which methods are to include the specified data members in their default implementation. `"DO_PRINT"` includes the member in the default implementation of the `psdisplay()` method. `"DO_START"` includes the member in the default implementation of the `start()` method, if applicable. `"DO_STOP"` includes the member in the default implementation of the `stop()` method, if applicable. `"DO_VOTE"` automatically registers the member with the `vmm_subenv::end_test` consensus instance, if applicable.

Multiple methods can be specified by adding or using the `or` symbolic values. All methods are specified by specifying the `"DO_ALL"` symbol.

### Example

*Example B-106*

```
`vmm_subenv_member_subenv(idler, DO_ALL - DO_STOP);
```

## vmm_subenv::end_test

End-of-test consensus interface.

### SystemVerilog

```
protected vmm_consensus end_test;
```

### OpenVera

```
protected vmm_consensus end_test;
```

### Description

Local copy of the `vmm_consensus` reference supplied to the constructor. It may be used to indicate if the sub-environment and its components consent to or oppose the ending of the test.

Unless an objection is indicated, the sub-environment will consent by default.

### Example

*Example B-107*

```
class my_vmm_subenv extends vmm_subenv;
   ...
   function new(string name,string inst,
       vmm_consensus end_test);
   super.new(name,inst,end_test);
      ...
   endfunction
   ...
endclass
```

## vmm_subenv::log

Message service interface for the sub-environment.

### SystemVerilog

```
vmm_log log;
```

### OpenVera

```
rvm_log log;
```

### Description

This property is set by the constructor, using the specified name and instance name. These names may be modified, afterward, using the `vmm_log::set_name()` or `vmm_log::set_instance()` methods.

### Example

*Example B-108*

```
class my_vmm_subenv extends vmm_subenv;
   vmm_log log;
   ...
   function new(string name,string inst,
          vmm_consensus end_test);
      ...
      `vmm_debug(log,"Sub Environment new done");
   endfunction
   ...
endclass
```

## vmm_subenv::new()

Creates a new instance of this sub-environment base class.

### SystemVerilog

```
function new(string name,
     string inst,
     vmm_consensus end_test,
    vmm_object parent = null);

With +define NO_VMM12
function new(string name,
             string inst,
             vmm_consensus end_test);
```

### OpenVera

```
task new(string name,
     string inst,
     vmm_consensus end_test);
```

### Description

Creates a new instance of this base class with the specified name and instance name. The specified name and instance names are used as the name and instance names of the log class property.

The specified end-of-test consensus object is assigned to the `end_test` class property, and may be used by the sub-environment to indicate that it opposes or consents to the ending of the test.

## Example

*Example B-109*

```
class my_vmm_subenv extends vmm_subenv;
   ...
   function new(string name,string inst,
         vmm_consensus end_test, vmm_object parent = null);
      super.new(name,inst,end_test, parent);
   endfunction
endclass
```

## vmm_subenv::report()

Reports information collected by the sub-environment.

### SystemVerilog

```
virtual function void report();
```

### OpenVera

```
virtual task report();
```

### Description

Reports status, coverage, or statistical information collected by the sub-environment, but not pass or fail of the test or sub-environment.

This method needs to be extended. It may also be invoked multiple times during the simulation.

### Example

*Example B-110*

```
class my_vmm_subenv extends vmm_subenv;
   ...
   virtual function void report()
      super.report();
      ...
   endfunction
   ...
endclass
```

## vmm_subenv::start()

Starts the sub-environment.

### SystemVerilog

```
virtual task start();
```

### OpenVera

```
virtual task start_t();
```

### Description

Starts the sub-environment. An error is reported, if this method is called before the sub-environment and DUT is reported as configured to the sub-environment base class, using the "vmm_consensus::unregister_voter()" method.

A stopped sub-environment may be restarted.

The base implementation must be called using the `super.start()` method, by any extension of this method in a user-defined extension of this base class.

### Example

*Example B-111*

```
class my_vmm_subenv extends vmm_subenv;
   ...
   virtual task start()
      super.start();
      this.my_xactor.start_xactor();
   endtask
endclass
```

## vmm_subenv::stop()

Stops the sub-environment.

### SystemVerilog

```
virtual task stop();
```

### OpenVera

```
virtual task stop_t();
```

### Description

Stops the sub-environment to terminate the test cleanly. An error is generated, if the sub-environment is not previously started.

The base implementation must be called using the `super.stop()` method, by any extension of this method in a user-defined extension of this base class.

### Example

*Example B-112*

```
class my_vmm_subenv extends vmm_subenv;
   ...
   virtual task stop()
      super.stop();
      this.my_xactor.stop_xactor();
      ...
   endtask
   ...
endclass
```

## ‘vmm_subenv_member_begin()

Starts of shorthand section.

### SystemVerilog

```
‘vmm_subenv_member_begin(class-name)
```

### OpenVera

Not supported.

### Description

Starts the shorthand section providing a default implementation for the `psdisplay()`, `start()` and `stop()` methods.

The class-name specified must be the name of the `vmm_subenv` extension class that is being implemented.

The shorthand section can only contain shorthand macros, and must be terminated by the “`‘vmm_subenv_member_end()`” method.

### Example

*Example B-113*

```
class tcpip_stack extends vmm_subenv;
   ...
   ‘vmm_subenv_member_begin(tcpip_stack)
      ...
   ‘vmm_subenv_member_end(tcpip_stack)
   ...
endclass
```

## ‘vmm_subenv_member_channel*()

Shorthand implementation for a channel data member.

### SystemVerilog

```
‘vmm_subenv_member_channel(member-name,
                           vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_channel_array(member-name,
                            vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_channel_aa_scalar(member-name,
                            vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_channel_aa_string(member-name,
                            vmm_subenv::do_what_e do_what)
```

### OpenVera

Not supported.

### Description

Adds the specified channel-type, array of channels, dynamic array of channels, scalar-indexed associative array of channels, or string-indexed associative array of channels data member to the default implementation of the methods specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by the `“‘vmm_subenv_member_begin()”` method.

## Example

*Example B-114*

```
class my_vmm_subenv extends vmm_subenv;
   data_channel subenv_channel;
   ...
   `vmm_subenv_member_begin(my_vmm_subenv)
      `vmm_subenv_member_channel(subenv_channel,DO_ALL)
      ...
   `vmm_subenv_member_end(my_vmm_subenv)
   ...
endclass
```

## ‘vmm_subenv_member_end()

End of shorthand section.

### SystemVerilog

```
‘vmm_subenv_member_end(class-name)
```

### OpenVera

Not supported.

### Description

Terminates the shorthand section providing a default implementation for the `psdisplay()`, `start()` and `stop()` methods.

The class-name specified must be the name of the `vmm_subenv` extension class that is being implemented.

The shorthand section must be started by the “`‘vmm_subenv_member_begin()`” method.

### Example

*Example B-115*

```
class my_vmm_subenv extends vmm_subenv;
   ...
   `vmm_subenv_member_begin(my_vmm_subenv)
      ...
   `vmm_subenv_member_end(my_vmm_subenv)
   ...
endclass
```

## ‘vmm_subenv_member_enum*()

Shorthand implementation for an enumerated data member.

### SystemVerilog

```
‘vmm_subenv_member_enum(member-name,
                        vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_enum_array(member-name,
                              vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_enum_aa_scalar(member-name,
                              vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_enum_aa_string(member-name,
                              vmm_subenv::do_what_e do_what)
```

### OpenVera

Not supported.

### Description

Adds the specified enum-type, array of enums, scalar-indexed associative array of enums, or string-indexed associative array of enums data member to the default implementation of the methods specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by the “‘vmm_subenv_member_begin()” method.

### Example

*Example B-116*

```
typedef enum {blue,green,red,black} my_colors;
```

```
class my_vmm_subenv extends vmm_subenv;
   my_colors  color;
   ...
   `vmm_subenv_member_begin(my_vmm_subenv)
      `vmm_subenv_member_enum(color,DO_ALL)
       ...
   `vmm_subenv_member_end(my_vmm_subenv)
   ...
endclass
```

## ‘vmm_subenv_member_scalar*()

Shorthand implementation for a scalar data member.

### SystemVerilog

```
‘vmm_subenv_member_scalar(member-name,
                          vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_scalar_array(member-name,
                                vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_scalar_aa_scalar(member-name,
                                vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_scalar_aa_string(member-name,
                                vmm_subenv::do_what_e do_what)
```

### OpenVera

Not supported.

### Description

Adds the specified scalar-type, array of scalars, scalar-indexed associative array of scalars or string-indexed associative array of scalars data member to the default implementation of the methods specified by the `’do_what’` argument.

A scalar is an integral type, such as bit, bit vector, and packed unions.

The shorthand implementation must be located in a section started by the `“‘vmm_subenv_member_begin()”` method.

## Example

*Example B-117*

```
class my_vmm_subenv extends vmm_subenv;
   bit [31:0] address;
   ...
   `vmm_subenv_member_begin(my_vmm_subenv)
      `vmm_subenv_member_scalar(address,DO_ALL)
       ...
   `vmm_subenv_member_end(my_vmm_subenv)
   ...
endclass
```

## ‘vmm_subenv_member_string*()

Shorthand implementation for a string data member.

### SystemVerilog

```
‘vmm_subenv_member_string(member-name,
                          vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_string_array(member-name,
                                vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_string_aa_scalar(member-name,
                                vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_string_aa_string(member-name,
                                vmm_subenv::do_what_e do_what)
```

### OpenVera

Not supported.

### Description

Adds the specified string-type, array of strings, scalar-indexed associative array of strings, or string-indexed associative array of strings data member to the default implementation of the methods specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by the “`‘vmm_subenv_member_begin()`” method.

### Example

*Example B-118*

```
class my_vmm_subenv extends vmm_subenv;
```

```
    string xactor_name;
    ...
    `vmm_subenv_member_begin(my_vmm_subenv)
       `vmm_subenv_member_string(xactor_name,DO_ALL)
        ...
    `vmm_subenv_member_end(my_vmm_subenv)
    ...
endclass
```

## ‘vmm_subenv_member_subenv*()

Shorthand implementation for a transactor data member.

### SystemVerilog

```
‘vmm_subenv_member_subenv(member-name,
                          vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_subenv_array(member-name,
                            vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_subenv_aa_scalar(member-name,
                            vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_subenv_aa_string(member-name,
                            vmm_subenv::do_what_e do_what)
```

### OpenVera

Not supported.

### Description

Adds the specified sub-environment-type, array of sub-environments, dynamic array of sub-environments, scalar-indexed associative array of sub-environments, or string-indexed associative array of sub-environments data member to the default implementation of the methods specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by the `“‘vmm_subenv_member_begin()”` method.

## Example

*Example B-119*

```
class sub_subenv extends vmm_subenv;
   function new(....);
      super.new(...);
      ...
   endfunction
endclass

class my_vmm_subenv extends vmm_subenv;
   sub_subenv sub_subenv_inst;
   ...
   `vmm_subenv_member_begin(my_vmm_subenv)
      `vmm_subenv_member_subenv(sub_subenv_inst,DO_ALL)
      ...
   `vmm_subenv_member_end(my_vmm_subenv)
   ...
endclass
```

## ‘vmm_subenv_member_user_defined()

User-defined shorthand implementation data member.

### SystemVerilog

```
‘vmm_subenv_member_user_defined(member-name)
```

### OpenVera

Not supported.

### Description

Adds the specified user-defined default implementation of the methods specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by the “‘vmm_subenv_member_begin()” method.

### Example

*Example B-120*

```
class my_vmm_subenv extends vmm_subenv;
   bit [7:0] subenv_id;
   ...
   `vmm_env_member_begin(my_vmm_subenv)
      `vmm_subenv_member_user_defined(subenv_id)
      ...
   `vmm_env_member_end(my_vmm_subenv)

  function bit do_subenv_id(vmm_subenv::do_what_e do_what)
     do_subenv_id = 1;
     case(do_what)
   endfunction
endclass
```

## ‘vmm_subenv_member_vmm_data*()

Shorthand implementation for a vmm_data-based data member.

### SystemVerilog

```
‘vmm_subenv_member_vmm_data(member-name,
                            vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_vmm_data_array(member-name,
                            vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_vmm_data_aa_scalar(member-name,
                            vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_vmm_data_aa_string(member-name,
                            vmm_subenv::do_what_e do_what)
```

### OpenVera

Not supported.

### Description

Adds the specified vmm_data-type, array of `vmm_datas`, scalar-indexed associative array of `vmm_datas`, or string-indexed associative array of `vmm_datas` data member to the default implementation of the methods specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by the `“‘vmm_subenv_member_begin()”` method.

## Example

*Example B-121*

```
class my_data extends vmm_data;
   ...
endclass

class my_vmm_subenv extends vmm_subenv;
   my_data   subenv_data;
   ...
   `vmm_subenv_member_begin(my_vmm_subenv)
      `vmm_subenv_member_vmm_data(subenv_data,DO_ALL)
      ...
   `vmm_subenv_member_end(my_vmm_subenv)
   ...
endclass
```

## ‘vmm_subenv_member_xactor*()

Shorthand implementation for a transactor data member.

### SystemVerilog

```
‘vmm_subenv_member_xactor(member-name,
                          vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_xactor_array(member-name,
                            vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_xactor_aa_scalar(member-name,
                            vmm_subenv::do_what_e do_what)

‘vmm_subenv_member_xactor_aa_string(member-name,
                            vmm_subenv::do_what_e do_what)
```

### OpenVera

Not supported.

### Description

Adds the specified transactor-type, array of transactors, dynamic array of transactors, scalar-indexed associative array of transactors, or string-indexed associative array of transactors data member to the default implementation of the methods specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by the `“‘vmm_subenv_member_begin()”` method.

## Example

*Example B-122*

```
class my_vmm_subenv extends vmm_subenv;
   data_gen subenv_xactor;
   ...
   `vmm_subenv_member_begin(my_vmm_subenv)
      `vmm_subenv_member_xactor(subenv_xactor,DO_ALL)
      ...
   `vmm_subenv_member_end(my_vmm_subenv)
   ...
endclass
```

# vmm_test

The `vmm_test` class is an extension of `vmm_timeline`, and handles the test execution timeline with all of the default predefined phases. This is used as the base class for all tests.

Instances of this class must be either root objects or children of `vmm_test` objects.

## Example

```
class my_test1 extends vmm_test;
   `vmm_typename(my_test1)
   function new(string name);
      super.new(name);
   endfunction

   function void config_ph;
      cfg cfg1 = new;
      if (cfg1.randomize)
         `vmm_note (log, "CFG randomized successfully" );
      else
         `vmm_error (log, "CFG randomization failed" );
   endfunction
endclass
```

## Summary


- vmm_test::get_doc() ..... page B-247
- vmm_test::get_name() ..... page B-248
- vmm_test::log ..... page B-249
- vmm_test::new() ..... page B-251
- vmm_test::run() ..... page B-252
- vmm_test::set_config() ..... page B-253
- `vmm_test_begin() ..... page B-254
- `vmm_test_end() ..... page B-256

## vmm_test::get_doc()

Returns the description of a test.

### SystemVerilog

```
virtual function string get_doc();
```

### OpenVera

Not supported.

### Description

Returns the short description of the test that was specified in the constructor.

### Example

*Example B-123*

```
class my_test extends vmm_test;
   function new();
      super.new("my_test");
   endfunction
   static my_test this_test = new();
   virtual task run(vmm_env env);
      `vmm_note(this.log,
         {"Running test ", this.get_doc()});
      ...
   endtask
endclass
```

## vmm_test::get_name()

Returns the name of a test.

### SystemVerilog

```
virtual function string get_name();
```

### OpenVera

Not supported.

### Description

Returns the name of the test that was specified in the constructor.

### Example

*Example B-124*

```
class my_test extends vmm_test;
   function new();
      super.new("my_test");
   endfunction
   static my_test this_test = new();
   virtual task run(vmm_env env);
      `vmm_note(this.log,
         {"Running test ", this.get_name()});
      ...
   endtask
endclass
```

## vmm_test::log

Message service interface for the testcase.

### SystemVerilog

```
vmm_log log;
```

### OpenVera

Not supported.

### Description

Message service interface instance that can be used to generate messages in the `vmm_test::run()` method.

The name of the message service interface is "Testcase", and the instance name is the name specified to the `vmm_test::new()` method.

### Example

*Example B-125*

```
program test;
   class test_100 extends vmm_test;
      vmm_env env;
      function new();
         super.new("test_100", "Single Read");
      endfunction
      task run(vmm_env env1);
         `vmm_note(log,"Test Started");
         $cast(env, env1);
      endtask
   endclass
```

```
   initial begin
      test_100 T;
      T = new;
      T.run(T.env);
   end
endprogram
```

## vmm_test::new()

Creates an instance of the testcase.

### SystemVerilog

```
function new(string name,
string doc = "",
vmm_object parent = null);
```

### OpenVera

Not supported.

### Description

Creates an instance of the testcase, its message service interface, and registers it in the global testcase registry under the specified name. A short description of the testcase may also be specified.

### Example

*Example B-126*

```
class my_test extends vmm_test;
   function new();
      super.new("my_test");
   endfunction
   static my_test this_test = new();
   virtual task run(vmm_env env);
      ...
   endtask
endclass
```

## vmm_test::run()

Runs a testcase.

### SystemVerilog

```
virtual task run(vmm_env env);
```

### OpenVera

Not supported.

### Description

The test itself.

The default implementation of this method calls `env.run()`. If a different test implementation is required, the default implementation of this method must not be invoked using the `super.run()` method.

This method should not call `vmm_log::report()`.

### Example

*Example B-127*

```
class my_test extends vmm_test;
   virtual task run(vmm_env env);
      tb_env my_env;
      $cast(my_env, env);
      my_env.start();
      my_env.gen[0].start_xactor();
      my_env.run();
   endtask
endclass
```

## vmm_test::set_config()

### SystemVerilog

```
virtual function void vmm_test::set_config()
```

### Description

This method may be used to set `vmm_unit` factory instances and configuration parameters in `vmm_unit` instances outside of the scope of the test module, using the `classname::override_with_*()` and `vmm_opts::set_*()` methods.

This method can only be used if tests are executed one per simulation. When this method is used, tests cannot be concatenated.

### Example

```
class my_ahb_trans extends vmm_object;
   ...
   `vmm_class_factory(my_ahb_trans)
endclass

class my_test1 extends vmm_test;
   `vmm_typename(my_test1)
   function new(string name);
      super.new(name);
   endfunction

   function set_config();
      ahb_trans::override_with_new("@%*",
         my_ahb_trans::this_type, log, `__FILE__,
         `__LINE__);
   endfunction
   ...
endclass
```

## ‘vmm_test_begin()

Shorthand macro to define a testcase class.

### SystemVerilog

```
‘vmm_test_begin(testclassname, envclassname, doc)
```

### OpenVera

Not supported.

### Description

Shorthand macro that may be used to define a user-defined testcase implemented using a class based on the `vmm_test` class. The first argument is the name of the testcase class that will also be used as the name of the testcase in the global testcase registry. The second argument is the name of the environment class that will be used to execute the testcase. A data member of that type named `"env"` will be defined and assigned, ready to be used. The third argument is a string, which is used to document the purpose of the test.

This macro can be used to create the testcase class up to and including the declaration of the `vmm_test::run()` method. This macro can then be followed by variable declarations and procedural statements. The instance of the verification environment of the specified type can be accessed as `"this.env"`. It must be preceded by any `import` statement required by the test implementation.

## Example

The following example shows how the testcase from Example B-126 and Example B-127 can be implemented, using shorthand macros.

*Example B-128*

```
import tb_env_pkg::*;

`vmm_test_begin(my_test, tb_env, "Simple test")
      this.env.build();
      this.env.gen[0].stop_xactor();
      this.env.run();
`vmm_test_end(my_test)
```

## ‘vmm_test_end()

Shorthand macro to define a testcase class.

### SystemVerilog

```
‘vmm_test_end(testclassname)
```

### OpenVera

Not supported.

### Description

Shorthand macro that may be used to define a user-defined testcase implemented using a class, based on the `vmm_test` class. The first argument must be the same name specified as the first argument of the `‘vmm_test_begin()` macro.

This macro can be used to end the testcase class, including the implementation of the `vmm_test::run()` method.

### Example

The following example shows how the testcase from Example B-126 and Example B-127 can be implemented, using shorthand macros.

*Example B-129*

```
‘vmm_test_begin(my_test, tb_env, “Simple test”)
      this.env.build();
      this.env.gen[0].stop_xactor();
      this.env.run();
‘vmm_test_end(my_test)
```

# vmm_test_registry

Global test registry that can be optionally used to implement runtime selection of tests.

No constructor is documented, because this class is implemented using a singleton pattern.  Its functionality is accessed strictly through static members.

## Summary


- vmm_test_registry::list() ..... page B-258
- vmm_test_registry::run() ..... page B-259

## vmm_test_registry::list()

Lists all available tests.

### SystemVerilog

```
static function void list();
```

### OpenVera

Not supported.

### Description

Lists the tests that are registered with the global test registry.

This method is invoked automatically by the `vmm_test_registry::run()` method, followed by a call to `$finish()`, if the `+vmm_test_help` option is specified.

### Example

*Example B-130*

```
program test;
   `include "test.lst"
   i2c_env env;

   initial begin
      vmm_test_registry registry = new;
      env = new;
      registry.list();
      registry.run(env);
   end
endprogram
```

## vmm_test_registry::run()

Runs a testcase.

### SystemVerilog

```
static task run(vmm_env env);
```

### OpenVera

Not supported.

### Description

Runs a testcase on the specified verification environment. Using SystemVerilog, this method must be invoked in a *program* thread to satisfy *Verification Methodology Manual* rules.

If more than one testcase is registered, then the name of a testcase must be specified using the `"+vmm_test"` runtime string option. For more information, see the section, `vmm_opts::get_string()` to know how to specify runtime string options. If only one test is registered, then it is run by default without having to specify its name at runtime.

A default testcase, named `"Default"` that simply invokes `env::run()`, is automatically available if no testcase is previously registered under that name.

### Example

*Example B-131*

```
program top;
   tb_env env = new();
```

```
   initial vmm_test_registry::run(env);
endprogram
```

## vmm_timeline

The `vmm_timeline` user-defined class coordinates simulation through a user-defined timeline, with predefined test phases as follows:

- build
- configure
- connect
- configure_test
- start_of_sim
- reset
- training
- config_dut
- start
- start_of_test
- run
- shutdown
- cleanup
- report
- final

Phases may be subsequently added or removed as needed.

## Summary


- vmm_timeline::abort_phase() ..... page B-263
- vmm_timeline::append_callback() ..... page B-264
- vmm_timeline::delete_phase() ..... page B-267
- vmm_timeline::display_phases() ..... page B-268
- vmm_timeline::get_current_phase_name() ..... page B-269
- vmm_timeline::get_next_phase_name() ..... page B-270
- vmm_timeline::get_phase() ..... page B-271
- vmm_timeline::get_previous_phase_name() ..... page B-272
- vmm_timeline::insert_phase() ..... page B-273
- vmm_timeline::jump_to_phase() ..... page B-275
- vmm_timeline::prepend_callback() ..... page B-276
- vmm_timeline::rename_phase() ..... page B-278
- vmm_timeline::reset_to_phase() ..... page B-279
- vmm_timeline::run_phase() ..... page B-280
- vmm_timeline::step_function_phase() ..... page B-281
- vmm_timeline::task_phase_timeout() ..... page B-282
- vmm_timeline::unregister_callback() ..... page B-283
- vmm_timeline_callbacks ..... page B-285
- vmm_timeline_callback::break_on_phase() ..... page B-286

## vmm_timeline::abort_phase()

Aborts the specified phase, if currently executing.

### SystemVerilog

```
function void abort_phase(string name,  string fname = "",
    int lineno = 0);
```

### Description

Aborts the execution of the specified phase, if it is the currently executing phase in the timeline. If another phase is executing, it generates a warning message if the specified phase is already executed to completion, and generates an error message if the specified phase is not yet started. The `fname` and `lineno` arguments are used to track the file name and the line number where this method is invoked from.

### Example

```
class test extends vmm_test;
   vmm_timeline topLevelTimeline;
endclass
...
initial begin
   test test1 = new ("test1", "test1");
   ...
   fork
      test1.topLevelTimeline.run_phase("reset");
      #(reset_cycle) test1.topLevelTimeline.abort_phase (
         "reset");
      ...
   join_any
   disable fork;
   ...
end
```

## vmm_timeline::append_callback()

Appends the specified callback.

### SystemVerilog

```
function void append_callback( vmm_timeline_callbacks cb);
```

### Description

Appends the specified callback extension to the callback registry, for this timeline. Returns true, if the registration was successful.

### Example

```
class timeline_callbacks extends vmm_timeline_callbacks;
   virtual function void my_f1();
   endfunction
endclass

class timelineExtension extends vmm_timeline;
   function new (string name, string inst,
         vmm_unit parent=null);
      super.new(name,inst,parent);
   endfunction

   function void build_ph();
      `vmm_callback(timeline_callbacks,my_f1());
   endfunction:build_ph
   ...
endclass

class timelineExtension_callbacks extends
      timeline_callbacks;
   int my_f1_counter++;
   virtual function void my_f1();
      my_f1_counter++;
   endfunction
```

```
endclass

initial begin
   timelineExtension tl = new ("my_timeline", "t1");
   timelineExtension_callbacks cb1 = new();
   tl.append_callback(cb1);
   ...
end
```

## vmm_timeline::configure_test_ph()

Configures the environment from testcase.

### SystemVerilog

```
function void configure_test_ph();
```

### Description

The `configure_test_ph` is the method that gets executed at the beginning of the test root timeline. The test-specific run-time configuration should be put in `configure_test_ph()` (options, callbacks, and so on). For multiple test concatenation, the default rollback for the tests in sequence is this `configure_test_ph`. Also, for multiple tests, the `configure_test phase` is run, even if timeline is not reset before it, followed by the test root timeline from reset point (set through `` `VMM_TEST_IS_CONCATENABLE `` macro) to the end (start_of_sim, reset, training, config_dut, run, shutdown, cleanup, and report) for the subsequent tests.

### Example

```
class test_read_back2back extends vmm_test;
  function new(string name);
     super.new(name);
  endfunction
  virtual function void configure_test_ph();
     test_read_back2back_test_trans tr = new();
     tr.address = 'habcd_1234;
     tr.address.rand_mode(0);
   cpu_trans::override_with_copy("@%*", tr, log, `__FILE__,
`__LINE__);
     vmm_opts::set_int("%*:num_scenarios", 50);
  endfunction
endclass
```

## vmm_timeline::delete_phase()

Deletes the specified phase from timeline.

### SystemVerilog

```
function bit delete_phase(string phase_name,
   string fname = "", int lineno = 0);
```

### Description

Deletes the specified phase in this timeline. Returns false, if the phase does not exist.

The `fname` and `lineno` arguments are used to track the file name and the line number where this method is invoked from.

### Example

```
class groupExtension extends vmm_group;
   function void build_ph ();
      vmm_timeline t = this.get_timeline();
      ...
      t.delete_phase ("connect");
      ...
   endfunction
endclass
```

## vmm_timeline::display_phases()

Displays all phases left to be executed.

### SystemVerilog

```
function void display_phases();
```

### Description

Displays all phases left to be executed, for this timeline.

### Example

```
class test extends vmm_test;
   ...
   user_timeline topLevelTimeline;
   ...
endclass
...
initial begin
   test test1 = new ("test1", "test1");
   ...
   fork
      begin
         test1.topLevelTimeline.run_phase();
      end
   begin
         #20 test1.topLevelTimeline.display_phases();
      end
      ...
   join
   ...
end
```

## vmm_timeline::get_current_phase_name()

Displays the current executing phase of the timeline.

### SystemVerilog

```
function string get_current_phase_name();
```

### Description

Displays the current phase, where the timeline phase execution is at a given point of time.

### Example

```
class test extends vmm_test;
   ...
   user_timeline topLevelTimeline;
   ...
endclass
...
initial begin
   test test1 = new ("test1", "test1");
   ...
   fork
      begin
         test1.topLevelTimeline.run_phase();
      end
      begin
         #20 `vmm_note (log, psprintf("Current Simulation
          Phase for test1 is : %s ",
          test1.topLevelTimeline.get_current_phase_name())
        );
      end
      ...
   join
   ...
end
```

## vmm_timeline::get_next_phase_name()

Returns the name of the following phase.

### SystemVerilog

```
function string get_next_phase_name(string name);
```

### Description

Returns the name of the phase that follows the specified phase. Returns $, if the specified phase is the last one. Returns ?, if the specified phase is unknown.

### Example

```
class groupExtension extends vmm_group;
   ...
   function void build_ph ();
      string nxt_ph;
      vmm_timeline t = this.get_timeline();
      ...
      nxt_ph = t.get_next_phase_name ("start_of_sim");
      //returns "reset"
      ...
   endfunction
endclass
```

## vmm_timeline::get_phase()

Returns the phase descriptor for a specified phase.

### SystemVerilog

```
function vmm_phase get_phase(string name);
```

### Description

Returns the descriptor of the specified phase in this timeline. Returns null if the specified phase is unknown.

### Example

```
class groupExtension extends vmm_group;
   ...
   function void build_ph();
      vmm_phase ph;
      vmm_timeline t = this.get_timeline();
      ...
      ph = t.get_phase ("start_of_sim");
      ...
   endfunction
endclass
```

## vmm_timeline::get_previous_phase_name()

Returns the name of the preceding phase.

### SystemVerilog

```
function string get_previous_phase_name(string name);
```

### Description

Returns the name of the phase that precedes the specified phase. Returns ^, if the specified phase is the first one. Returns ?, if the specified phase is unknown.

### Example

```
class groupExtension extends vmm_group;
   ...
   function void build_ph ();
      string prv_ph;
      vmm_timeline t = this.get_timeline();
      ...
      prv_ph = t.get_previous_phase_name ("start_of_sim");
         //returns "configure_test"
      ...
   endfunction
endclass
```

## vmm_timeline::insert_phase()

Inserts a phase in timeline.

### SystemVerilog

```
function bit insert_phase(string phase_name,
   string before_name, vmm_phase_def def, string fname = "",
   int lineno = 0);
```

### Description

Creates the specified phase (`phase_name`) before the specified phase (`before_name`) in this timeline, and issues a note that a new user-defined phase is defined. The argument `def` specifies the phase instance to be inserted. If the phase already exists, adds this definition to the existing phase definition. If the `before_name` is specified as a caret (^), then inserts the phase at the beginning of the timeline. If it is specified as a dollar sign ($), then inserts the phase at the end of the timeline. Returns true, if the phase insertion was successful.

The `fname` and `lineno` arguments are used to track the file name and the line number where this method is invoked from.

### Example

```
typedef class groupExtension
class udf_start_def extends vmm_fork_task_phase_def
      #(groupExtension);
   ...
endclass
class groupExtension extends vmm_group;
   ...
   function void build_ph ();
      vmm_timeline t = this.get_timeline();
```

```
      udf_start_def udfstartph = new;
      ...
      if(t.insert_phase("udf_start", "start_of_sim",
            udfstartph) == 0)
         `vmm_error (log, " ... ");
   endfunction
endclass
```

## vmm_timeline::jump_to_phase()

Aborts the execution of the timeline immediately and jump to the beginning of the specified phase.

### SystemVerilog

```
function void jump_to_phase(string name,string fname = "",
int lineno = 0);
```

### Description

Aborts the execution of the timeline, and immediately jumps to the beginning of the specified phase (but does not start executing it). Generates a warning message, if the specified phase is already started or completed.

Executing a phase without the intervening phases may cause severe damage to the state of the executing testcase and verification environment, and should be used with care. You should typically use to abort a testcase or simulation, and jump to the report phase. The `fname` and `lineno` arguments are used to track the file name and the line number where this method is invoked from.

### Example

```
class timelineExtension #(string jump_phase = "report",
      int delay_in_jump = 10) extends vmm_timeline;
   ...
   task reset_ph;
      #delay_in_jump jump_to_phase(jump_phase);
   endtask
   ...
endclass
```

## vmm_timeline::prepend_callback()

Prepends the specified callback.

### SystemVerilog

```
function void prepend_callback(vmm_timeline_callbacks cb);
```

### Description

Prepends the specified callback extension to the callback registry, for this timeline. Returns true, if the registration was successful.

### Example

```
class timeline_callbacks extends vmm_timeline_callbacks;
   virtual function void my_f1();
   endfunction
endclass

class timelineExtension extends vmm_timeline;
   function new (string name, string inst,
         vmm_unit parent=null);
      super.new(name,inst,parent);
   endfunction

   function void build_ph();
      `vmm_callback(timeline_callbacks,my_f1());
   endfunction:build_ph
   ...
endclass

class timelineExtension_callbacks extends
      timeline_callbacks;
   int my_f1_counter++;
   virtual function void my_f1();
      my_f1_counter++
   endfunction
```

```
endclass

initial begin
   timelineExtension tl = new ("my_timeline", "t1");
   timelineExtension_callbacks cb1 = new();
   timelineExtension_callbacks cb2 = new();
   tl.append_callback(cb1);
   tl.prepend_callback(cb2);
   ...
end
```

## vmm_timeline::rename_phase()

Provides a new name to the specified phase.

### SystemVerilog

```
function bit rename_phase(string old_name, string new_name,
string fname = "", int lineno = 0);
```

### Description

Renames the specified phase `old_name` in this timeline, to the new phase name `new_name`. Returns false, if the original named phase does not exist, or if a phase already exists with the new name. Generates a warning that a phase is renamed. Renaming timeline default phases is not allowed. The `fname` and `lineno` arguments are used to track the file name and the line number where this method is invoked from.

### Example

```
class groupExtension extends vmm_group;
   ...
   function void build_ph ();
      vmm_timeline t = this.get_timeline();
      ...
      // Renaming predefined phase 'start_of_sim'
      if(t.rename_phase("start_of_sim",
            "renamed_start_of_sim") == 0)
         `vmm_error(log, " ... ");
      ...
   endfunction
endclass
```

## vmm_timeline::reset_to_phase()

Resets timeline to the specified phase.

### SystemVerilog

```
function void reset_to_phase(string name, string fname="",
int lineno=0);
```

### Description

Resets this timeline to the specified phase `name`. Any task-based phase, which is concurrently running is aborted. If the timeline is reset to the `configure` phase or earlier, all of its `vmm_unit` sub-instances are enabled, along with itself.

The `fname` and `lineno` arguments are used to track the file name and the line number where this method is invoked from.

### Example

```
class test extends vmm_test;
   user_timeline topLevelTimeline;
endclass
...
initial begin
   test test1 = new ("test1", "test1");
   fork
      test1.topLevelTimeline.run_phase();
      //Assume topLevelTimeline is  going to run more
      //than #9 delay
      #9 test1.topLevelTimeline.reset_to_phase ("build");
   join
end
```

## vmm_timeline::run_phase()

Runs a timeline, up to and including the specified phase.

### SystemVerilog

```
task run_phase(string name = "$", string fname = "", int
lineno = 0);
```

### Description

Executes the phases in this timeline, up to and including the specified phase by argument `name`. For name $, run all phases.

The `fname` and `lineno` arguments are used to track the file name and the line number where this method is invoked from.

### Example

```
class test extends vmm_test;
   ...
   vmm_timeline topLevelTimeline;
   ...
endclass
...
initial begin
   test test1 = new ("test1", "test1");
   test1.topLevelTimeline.run_phase ("build");
   ...
   test1.topLevelTimeline.run_phase ();
end
```

## vmm_timeline::step_function_phase()

Steps to the next executable phase.

### SystemVerilog

```
function void step_function_phase(string name,
string fname = "", int lineno = 0);
```

### Description

Executes the specified function phase in this timeline. Must be a function phase, and must be the next executable phase. The `fname` and `lineno` arguments are used to track the file name and the line number where this method is invoked from.

### Example

```
class test extends vmm_test;
   ...
   vmm_timeline topLevelTimeline;
   ...
endclass
...
initial begin
   test test1 = new ("test1", "test1");
   ...
   test1.topLevelTimeline.run_phase ("configure");
   test1.topLevelTimeline.step_function_phase ("connect");
   test1.topLevelTimeline.step_function_phase (
      "configure_test");
   ...
end
```

## vmm_timeline::task_phase_timeout()

Sets the timeout value for any task phase.

### SystemVerilog

```
function bit task_phase_timeout(string name,
   int unsigned delta, vmm_log::severities_e
   error_severity=vmm_log::ERROR_SEV, string fname = "",
   int lineno = 0);
```

### Description

Sets the timeout value - as specified by `delta` - for the completion of the specified task phase. If the task phase does not complete within the time specified in the timeout value, then an error message is generated. Message severity, which is error by default, can be overridden using the `error_severity` argument.Returns false, if the specified phase does not exist or is not a task phase.

A timeout value of 0 specifies no timeout value. Calling this method, while the phase is currently executing, causes the timer to be reset to the specified value. By default, phases do not have timeouts. The `fname` and `lineno` arguments are used to track the file name and the line number where this method is invoked from.

### Example

```
class groupExtension extends vmm_group;
   function void build_ph ();
      vmm_timeline t = this.get_timeline();
      if(t.task_phase_timeout("reset",4) == 0)
         `vmm_error (log, " ... ");
      ...
   endfunction
endclass
```

## vmm_timeline::unregister_callback()

Unregisters a callback.

### SystemVerilog

```
function void unregister_callback(
vmm_timeline_callbacks cb);
```

### Description

Removes the specified callback extension from the callback registry, for this timeline. Returns true, if the unregistration was successful.

### Example

```
class timeline_callbacks extends vmm_timeline_callbacks;
   virtual function void my_f1();
   endfunction
endclass

class timelineExtension extends vmm_timeline;
   function new (string name, string inst, vmm_unit
         parent=null);
      super.new(name,inst,parent);
   endfunction

   function void build_ph();
      `vmm_callback(timeline_callbacks,my_f1());
   endfunction:build_ph
   ...
endclass

class  timelineExtension_callbacks extends
      timeline_callbacks;
   int my_f1_counter++;
   virtual function void my_f1();
      my_f1_counter++;
```

```
   endfunction
endclass

initial begin
   timelineExtension tl = new ("my_timeline", "t1");
   timelineExtension_callbacks cb1 = new();
   timelineExtension_callbacks cb2 = new();
   tl.append_callback(cb1);
   tl.append_callback(cb2);
   ...
   tl.unregister_callback(cb2);
   ...
end
```

# vmm_timeline_callbacks

Facade class for callback methods provided by a timeline.

## Example

```
class timeline_callbacks extends vmm_timeline_callbacks;
   virtual function void my_f1();
   endfunction
   virtual function void my_f2();
   endfunction
endclass
```

## Summary


- vmm_timeline_callback::break_on_phase() ..... page B-286

## vmm_timeline_callback::break_on_phase()

This method is called, if the `+break_on_X_phase` option is set for this timeline instance.

### SystemVerilog

```
function void vmm_timeline_callbacks::break_on_phase(
   vmm_timeline t1, string name)
```

### Description

This method is called, if the `+break_on_X_phase` option is set for this timeline instance. The arguments are the instance of the timeline and the name of the phase (`X`). If no callbacks are registered, `$stop` is called instead of this method.

### Example

```
class timeline_callbacks extends vmm_timeline_callbacks;
   vmm_log log;
   function new(vmm_log log);
      this.log = log;
   endfunction

   function void break_on_phase(vmm_timeline tl,
         string name);
      if(name=="reset")
         `vmm_note(log,
            "user callback executing for reset phase");
   endfunction
endclass

vmm_timeline tl;

initial begin
   timeline_callbacks  cb1;
```

```
   tl = new("my_timeline", "tl");
   cb1 = new(tl.log);
   tl.append_callback(cb1);
   tl.run_phase();
end
```

# vmm_tlm

This class contains the `sync_e` enumerated for various phases of the transaction. All TLM port classes use this enumerated value as the default template for defining the phases of the transaction.

## SystemVerilog

```
class vmm_tlm;
   typedef enum { TLM_REFUSED, TLM_ACCEPTED,
      TLM_UPDATED, TLM_COMPLETED } sync_e;
   typdef enum {BEGIN_REQ, END_REQ, BEGIN_RESP,
      END_RESP} phase_e;
   typedef enum { TLM_BLOCKING_PORT, TLM_BLOCKING_EXPORT,
      TLM_NONBLOCKING_FW_PORT, TLM_NONBLOCKING_FW_EXPORT,
      TLM_NONBLOCKING_PORT,TLM_NONBLOCKING_EXPORT,
      TLM_ANALYSIS_PORT,TLM_ANALYSIS_EXPORT} intf_e;
   sync_e sync;
endclass
```

## Description

This class provides enumerated type `sync_e`, which is the response status from a non-blocking transport function call, upon receiving a transaction object.

The enumerated type `phase_e` contains various phases of a transaction object. These phases can be updated by different components that access the same transaction object.

The enumerated type `intf_e` is used to connect the `vmm_channel_typed` to TLM transport ports, TLM transport exports, and TLM analysis ports and exports.

The `vmm_tlm` class also provides static methods to print, check, and report the bindings of all TLM ports and exports, under a specified root.

## Summary


- `vmm_tlm::check_bindings()` ..... page B-297
- `vmm_tlm::print_bindings()` ..... page B-298
- `vmm_tlm::report_unbound()` ..... page B-299

# vmm_tlm_extension_base

Generic payload extensions base class. This class must be extended to define user extensions of the `vmm_tlm_generic_payload` class.

## SystemVerilog

```
class vmm_tlm_extension_base extends vmm_data;
```

## Description

This class is used to define extensions of the `vmm_tlm_generic_payload` class.

# vmm_tlm_generic_payload

This data class contains attributes, as defined by the OSCI TLM2.0 `tlm_generic_payload` class. The class is extended from the `vmm_rw_access` class, which is in turn extended from `vmm_data` class. The SystemVerilog implementation uses the VMM data shorthand macros, to implement all methods that are implemented by the `vmm_data` class.

Generic payload class can be extended to have user defined functionality by extending `vmm_tlm_extension_base`. The `vmm_tlm_generic_payload` class has a dynamic array of `vmm_tlm_extension_base`, which is used to store the user extensions.

## SystemVerilog

```
class vmm_tlm_generic_payload extend vmm_rw_access;
   typedef enum {TLM_READ_COMMAND = 0,
                 TLM_WRITE_COMMAND = 1,
                 TLM_IGNORE_COMMAND = 2
                }tlm_command;

   typedef enum {TLM_OK_RESPONSE = 1,
                 TLM_INCOMPLETE_RESPONSE = 0,
                 TLM_GENERIC_ERROR_RESPONSE = -1,
                 TLM_ADDRESS_ERROR_RESPONSE = -2,
                 TLM_COMMAND_ERROR_RESPONSE = -3,
                 TLM_BURST_ERROR_RESPONSE = -4,
                 TLM_BYTE_ENABLE_ERROR_RESPONSE = -5
                 }tlm_response_status;

   rand longint              m_address;
   rand tlm_command          m_command;
   rand byte                 m_data[];
   rand int unsigned         m_length;
        tlm_response_status  m_response_status;
```

```
      bit                   m_dmi_allowed = 0;
   rand byte                m_byte_enable[];
   rand int unsigned        m_byte_enable_length;
   rand int unsigned        m_streaming_width;
        int unsigned        min_m_length;
        int unsigned        max_m_length;
        int unsigned        max_m_byte_enable_length;

   constraint c_length_valid
     { m_data.size == m_length;
       m_length>min_m_length;
     }
   constraint c_data_size_reasonable
     {m_length<=max_m_length;
     }

   constraint c_byte_enable_valid
     { m_byte_enable.size == m_byte_enable_length;
     }
   constraint c_byte_enable_size_reasonable
     { m_byte_enable_length<=max_m_byte_enable_length;
     }

endclass: vmm_tlm_generic_payload
```

## Description

The class members are kept public to access methods to set and get the members that are not provided. The DMI and Debug Interfaces are not part of the VMM-TLM implementation, and therefore not included in the `vmm_tlm_generic_payload` class.

The `m_data` and `m_data_enable` values are constrained to small values of 16 and 256, respectively for better performance. If values larger than these are required, then the constraint blocks such as `c_data_size_reasonable` and `c_byte_enable_size_reasonable` should be switched off and applicable ranges can be provided.

For more details on the attributes of the `vmm_tlm_generic_payload` class, refer to the OSCI TLM-2.0 User Guide.

## Summary


- `vmm_tlm_generic_payload::set_extensions()` ..... page B-294
- `vmm_tlm_generic_payload::get_extensions()` ..... page B-295
- `vmm_tlm_generic_payload::clear_extensions()` ..... page B-296

## vmm_tlm_generic_payload::set_extensions()

To add user-defined extension to `vmm_tlm_extension_base` class array in the generic payload class.

### SystemVerilog

```
function vmm_tlm_extension_base set_extension(int
index, vmm_tlm_extension_base ext);
```

### Description

This function is used to assign the extension base to the dynamic array in the generic payload class at the specified index, and returns the old extension at that index.

## vmm_tlm_generic_payload::get_extensions()

Returns the user-defined extension at the specified index from the extensions array of the generic payload class.

### SystemVerilog

```
function vmm_tlm_extension_base get_extension(int
index);
```

### Description

This function is used to get the extension from the dynamic array in the generic payload class in that index.

## vmm_tlm_generic_payload::clear_extensions()

To clear the user-defined extension at the specified index from the extensions array in the generic payload class.

### SystemVerilog

```
function void clear_extension(int index);
```

### Description

This function is used to clear the extension from the dynamic array in the generic payload class in that index.

### Example

```
class my_extensions extend vmm_tlm_extension_base;
   rand int data32;
   rand bit[7:0] data8;
end class

class producer extends vmm_xactor;
   vmm_tlm_nb_transport_port#(producer) nb_port;
   task run_ph();
      my_data tr;
       my_extensions tr_ex, temp_tr_ext;
      while(1) begin
         tr = new();
          tr_ex = new();
         tr.set_extensions(0,tr_ex);
         temp_tr_ext = tr.get_extensions(0);
               this.nb_port.nb_tranport_fw(tr,ph,delay);
               tr.clear_extensions(0);
         #5;
      end
   endtask
endclass
```

## vmm_tlm::check_bindings()

Static method to check if minimum bindings exist for all TLM ports and exports under the specified root.

### SystemVerilog

```
static function check_bindings(vmm_object root= null);
```

### Description

A warning is generated if a port is unbound ,or if an export contains less than the minimum bindings specified for the export. Analysis port bindings are reported with debug severity. If root is not specified, then the binding checks are done for all TLM ports and exports in the environment.

The `check_bindings()` method is also available with all TLM ports and exportsm and can be invoked for the particular port object.

### Example

```
class my_env extends vmm_group;
   function void start_of_sim_ph();
      ...
      vmm_tlm::check_bindings(this);
   endfunction
endclass
```

## vmm_tlm::print_bindings()

Static method used to print the bindings of all TLM ports and exports, instantiated under a specified root.

### SystemVerilog

```
static function print_bindings(vmm_object root = null);
```

### Description

Prints the bindings of all TLM ports and exports, including transport ports and exports, sockets and analysis ports, and exports instantiated under the `vmm_object,` specified by the root argument. If null is passed, then the bindings are printed for all TLM ports and exports in the environment.

The `print_bindings()` method is also available with all TLM ports and exports, and can be invoked for the particular port object.

### Example

```
class my_env extends vmm_group;
   function void start_of_sim_ph();
      ...
      vmm_tlm::print_bindings(this);
   endfunction
endclass
```

## vmm_tlm::report_unbound()

Static method to report all unbound TLM ports and to export instances available under a specified root.

### SystemVerilog

```
static function report_bindings(vmm_object root = null);
```

### Description

Reports all unbound TLM ports and exports, including transport ports and exports, sockets and analysis ports, and exports instantiated under the `vmm_object,` specified by the root argument. If null is passed, then the bindings are printed for all TLM ports and exports in the environment.

A warning is generated, if any TLM port or export under the specified root is left unbound. For analysis ports, a message with debug severity is generated.

The `report_unbound()` method is also available with all TLM ports and exports, and can be invoked for the particular port object.

### Example

```
class my_env extends vmm_group;
   function void start_of_sim_ph();
      ...
      vmm_tlm::report_unbound(this);
   endfunction
endclass
```

# vmm_tlm_analysis_port#(I,D)

Analysis ports are useful to broadcast transactions, to observers like scoreboards and functional coverage models. Analysis ports can be bound to any number of observers, through the observers analysis export.

The analysis port calls the write method of all the observers bound to it.

## SystemVerilog

```
class vmm_tlm_analysis_port#(
   type INITIATOR = vmm_tlm_xactor, type DATA = vmm_data,)
   extends vmm_tlm_analysis_port_base#(DATA);
```

## Description

The analysis port can be instantiated in any transactor class that wishes to broadcast the transaction object to the connected observers.

Any number of bindings are allowed for the analysis port. The analysis port calls the write methods of the connected analysis exports, which in turn execute the write methods of their respective parent components.

The `vmm_tlm_analysis_port_base` provides all the access methods that are provided by the `vmm_tlm_port_base` class. The methods provided by the `vmm_tlm_analysis_port_base` class are `get_peers()`, `get_n_peers()`, `get_peer_id()`, `get_peer()`, `tlm_bind()`, `tlm_unbind()`, and `tlm_import()`. For more information on these access methods, refer to the description provided in the `vmm_tlm_port_base` class.

## Example

```
class consumer extends vmm_xactor;
   vmm_tlm_analysis_port#(consumer) analysis_port =
         new(this,"consumer_analysis");

   function b_transport(int id=-1,my_trans trans,
                        ref int delay);
      this.analysis_port.write(trans);
   endfunction
endclass
```

# vmm_tlm_analysis_export#(T,D)

Analysis exports are used by observer components that implement a write method to receive broadcast transactions from other components that instantiate the `vmm_tlm_analysis_port` class. Analysis exports can be bound to any number of analysis ports, as specified in the constructor of the analysis export. The different analysis ports connected to this export can be distinguished using the peer identity of the analysis port.

The analysis export implements the write method, which is called by the analysis ports that are bound to this export.

## SystemVerilog

```
class vmm_tlm_analysis_export#(type T = vmm_tlm_xactor,
   type D = vmm_data)
   extends vmm_tlm_analysis_export_base#(D);
```

## Description

The analysis export can be instantiated in a component class that wishes to receive broadcast transaction objects from other components.

The `vmm_tlm_analysis_export_base` provides all access methods that are provided by the `vmm_tlm_export_base` class. The methods provided by `vmm_tlm_analysis_port_base` are:

```
get_peers()
get_n_peers()
get_peer_id()
get_peer()
tlm_bind()
bind_peer()
tlm_unbind()
```

```
unbind_peer()
tlm_import()
print_bindings()
check_bindings()
report_unbound()
```

For more information on these access methods, refer to the description provided in the `vmm_tlm_export_base` class. Methods to get and set the minimum and maximum bindings for the port are also provided.

Available methods are:

```
function vmm_tlm_analysis_export::set_max_bindings(
   int unsigned max);
function vmm_tlm_analysis_export::set_min_bindings(
   int unsigned min);
function int unsigned
   vmm_tlm_analysis_export::get_max_bindings();
function int unsigned
   vmm_tlm_analysis_export::get_min_bindings();
```

## Example

```
class scoreboard extends vmm_group;
   vmm_tlm_analysis_export#(scoreboard) analysis_export =
      new(this,"scb_analysis");
   function write(int id=-1, my_trans trans);
   endfunction
endclass
```

# ‘vmm_tlm_analysis_export(SUFFIX)

Shorthand macro to create unique class names of the analysis export. This is used if multiple `vmm_tlm_analysis_export` instances are required in the same observer class, each having its own implementation of the write method.

## SystemVerilog

```
`vmm_tlm_analysis_export(SUFFIX)
```

## Description

The use model is similar to the shorthand macros provided for the unidirectional exports. For more information, refer to the macro description of `` `vmm_tlm_nb_transport_fw_export ``.

## Example

```
class scoreboard extends vmm_group;
   `vmm_tlm_analysis_export(_1)
   `vmm_tlm_analysis_export(_2)
   vmm_tlm_analysis_export_1#(scoreboard) scb1;
   vmm_tlm_analysis_export_2#(scoreboard) scb2;
   function write_1 (int id=-1,my_trans trans);
      `vmm_note(log, $psprintf("Received %s from %0d",
        Trans.psdisplay(""), id);
   endfunction
   function write_2 (int id=-1,my_trans trans);
      `vmm_note(log, $psprintf("Received %s from %0d",
        Trans.psdisplay(""), id);
   endfunction
 endclass
```

# vmm_tlm_b_transport_export#(T,D)

Blocking transport export class.

Any class instantiating this blocking transport export, must provide an implementation of the `b_transport()` task.

## SystemVerilog

```
class vmm_tlm_b_transport_export#(
   type TARGET = vmm_tlm_xactor,
   type DATA = vmm_data)
   extends vmm_tlm_export_base#(DATA);
```

## Description

Class providing the blocking transport export. The parameter type `TARGET` is the class that instantiates the transport export. This defaults to `vmm_tlm_xactor`. The parameter `DATA` is the data type of the transaction the export services. The default is `vmm_data`.

The export can be bound to multiple ports, up to the maximum bindings, specified in the constructor of this class.

## Summary


- `vmm_tlm_b_transport_export() ..... page B-306
- vmm_tlm_b_transport_export::b_transport() ..... page B-308
- vmm_tlm_b_transport_export::new() ..... page B-309

## `vmm_tlm_b_transport_export()

Shorthand macro to create unique blocking transport exports. This is required if more than one export is bound in a target transactor.

### SystemVerilog

```
`vmm_tlm_b_transport_export(SUFFIX)
```

### Description

This macro creates a uniquified `vmm_tlm_b_transport_export` class, with the *`SUFFIX`* appended to the class name `vmm_tlm_b_transport_export`. The class with the name `vmm_tlm_b_transport_export`*`SUFFIX`* is created in the scope, where the macro is called.

This macro is required if there are multiple instances of the `vmm_tlm_b_transport_export,` and each requires a unique implementation of the `b_transport()` task in the parent transactor.

The `b_transport()` methods in the parent transactor must be uniquified using the same *`SUFFIX`* to `b_transport`.

Alternatively, if multiple ports need to service the parent transactor, then a single export with multiple bindings using unique ids can be used in place of the macro. The single `b_transport()` method can be programmed to serve the various ports depending on the id.

### Example

```
class consumer extends vmm_xactor;
   `vmm_tlm_b_transport_export(_1)
```

```
   `vmm_tlm_b_transport_export(_2)
   vmm_tlm_b_transport_export_1#(consumer) b_export1 =
      new(this, "export1");

   vmm_tlm_b_transport_export_2#(consumer) b_export2 =
      new(this,"export2");

   task b_transport_1(int id = -1, vmm_data trans,
                      ref int delay );
      trans.display("From export1");
   endtask

   task b_transport_2(int id = -1, vmm_data trans,
                      ref int delay);
      trans.display("From export2");
   endtask
endclass

class producer extends vmm_xactor;
   vmm_tlm_b_transport_port#(producer) b_port;
endclass

class my_env extends vmm_group;
   producer p1,p2;
   consumer c1;
   function void connect_ph();
      c1.b_export1.tlm_bind(p1.b_port);
      c1.b_export2.tlm_bind(p2.b_port);
   endfunction
endclass
```

## vmm_tlm_b_transport_export::b_transport()

Blocking transport method of the export.

### SystemVerilog

```
task b_transport(int id = -1, DATA trans, ref int delay );
```

### Description

Blocking transport task of the transport export. This task is internally called by the bound transport port. This task calls the `b_transport()` method of the parent transactor in which it is instantiated.

The specified `trans` argument is a handle of the transaction object, `id` specifies the binding identifier of this export, `delay` argument is the timing annotation.

### Example

```
class consumer extends vmm_xactor;
   vmm_tlm_b_transport_export#(consumer) b_export;
   task b_transport(int id = -1, vmm_data trans,
                    ref int delay);
      trans.display("From consumer");
   endtask
endclass
```

## vmm_tlm_b_transport_export::new()

Constructor of blocking transport export class.

### SystemVerilog

```
function new(TARGET parent, string name, int max_binds = 1 ,
   int min_binds = 0);
```

### Description

Sets the parent and instance name of the blocking transport export. Sets the maximum and minimum bindings allowed for this export. The default value of maximum bindings is 1, and the minimum binding is 0. An error is generated during `tlm_bind()`, if the current binding exceeds the maximum allowed bindings for the export. An error is generated during elaboration, if the export does not contain the minimum number of specified bindings.

### Example

```
class consumer extends vmm_xactor;
   vmm_tlm_b_transport_export#(consumer) b_export;
   function void build_ph();
      this.b_export = new(this,"consumer export",5,1);
   endfunction
endclass
```

# vmm_tlm_b_transport_port #(I,D)

Base class for modeling a blocking transport port.

## SystemVerilog

```
class vmm_tlm_b_transport_port #(
   type INITIATOR = vmm_tlm_xactor, type DATA = vmm_data)
extends vmm_tlm_port_base#(DATA);
```

## Description

Class providing the blocking transport port. The parameter type `INITIATOR` is the class that instantiates the transport port. This defaults to `vmm_tlm_xactor`. The parameter `DATA` is the data type of the transaction port services. The default is `vmm_data`.

The port can be bound to one export. A warning is generated, if the port is left unbound.

There is no backward path for the blocking transport.

## Summary


- vmm_tlm_b_transport_port::b_transport() ..... page B-311
- vmm_tlm_b_transport_port::new() ..... page B-312

## vmm_tlm_b_transport_port::b_transport()

TLM task for blocking transport.

### SystemVerilog

```
task b_transport(DATA trans, ref int delay);
```

### Description

TLM task for blocking transport. Invokes the `b_transport()` method of the bounded export. The `index` argument can be used for associating the `b_transport` call with the caller, this can be usefull for the target to identify which producers called this task. The `trans` argument is a handle of the transaction object. The `delay` argument is the timing annotation.

### Example

```
class producer extends vmm_xactor;
   vmm_tlm_b_transport_port#(producer) b_port;
   task run_ph();
      my_data tr;
      while(1) begin
         tr = new();
         this.b_port.b_tranport(tr, delay);
         $display("Transaction Completed");
      end
   endtask
endclass
```

## vmm_tlm_b_transport_port::new()

Constructor for blocking transport port class.

### SystemVerilog

```
function new(INITIATOR parent, string name);
```

### Description

Sets the parent and instance name of the blocking transport port.

### Example

```
class producer extends vmm_xactor;
   vmm_tlm_b_transport_port#(producer) b_port;
   function void build_ph();
      this.b_port = new(this,"producer port");
   endfunction
endclass
```

# vmm_tlm_export_base #(D,P)

Abstract base class for all TLM2.0 transport exports. This class contain the methods that are required by all TLM2.0 transport export implementations. Any user-defined export must be extended from this base class.

The parameter DATA is the type of the transaction object of the export services. The default type is `vmm_data`. The parameter PHASE is the type of the phasing class. The default value is `vmm_tlm::phase_e`.

## SystemVerilog

```
virtual class vmm_tlm_export_base #(type DATA = vmm_data,
   type PHASE = vmm_tlm::phase_e) extends vmm_tlm_base;
```

## Description

Sets the parent, if it is an extension of `vmm_object`. Sets the name of the instance.

## Summary


- vmm_tlm_export_base::get_n_peers() Function ..... page B-314
- vmm_tlm_export_base::get_peer() ..... page B-315
- vmm_tlm_export_base::get_peer_id() ..... page B-316
- vmm_tlm_export_base::get_peers() ..... page B-317
- vmm_tlm_export_base::new() ..... page B-318
- vmm_tlm_export_base::tlm_bind() ..... page B-319
- vmm_tlm_export_base::tlm_import() ..... page B-321
- vmm_tlm_export_base::tlm_unbind() ..... page B-323

## vmm_tlm_export_base::get_n_peers() Function

Returns the number of export bindings.

### SystemVerilog

```
function int get_n_peers();
```

### Description

Returns the number of port export bindings, as set with the `tlm_bind()` method.

### Example

```
class consumer extends vmm_xactor;
   vmm_tlm_b_transport_export#(consumer) b_export;
   function display_n_connections();
      $display("Export has %d bindings",
         this.b_export.get_n_peers());
   endfunction
endclass
```

## vmm_tlm_export_base::get_peer()

Returns the binding for the port.

### SystemVerilog

```
function vmm_tlm_port_base#(DATA,PHASE) get_peer(int
id = -1);
```

### Description

Returns the port bound to the current export, with the specified `id`. Null is returned, if the port does not have a binding with the specified `id`. If only one binding exists for the export, then the handle to be binding is returned without considering the `id` value passed.

### Example

```
class consumer extends vmm_xactor;
   vmm_tlm_b_transport_export#(consumer) b_export;
   function display_my_id();
      vmm_tlm_export_base peer;
      peer = this.b_export.get_peer(0);
      $display("My id = %d",peer.get_peer_id();
   endfunction
endclass
```

## vmm_tlm_export_base::get_peer_id()

Returns the `id` of this port, for its binding.

### SystemVerilog

```
function int get_peer_id(vmm_tlm_port_base#(DATA,PHASE)
   peer);
```

### Description

Returns the binding `id` of the specified port bound to this export. If the specified port is not bound to this export, then -1 is returned.

### Example

```
class my_env extends vmm_group;
   producer p1,p2;
   consumer c1;

   function void connect_ph();
      p1.b_port.tlm_bind(c1.b_export);
      p2.b_port.tlm_bind(c1.b_export);
      int p1_id = c1.b_export.get_peer_id(p1.b_port);
      int p2_id = c1.b_export.get_peer_id(p2.b_port);
   endfunction
endclass
```

## vmm_tlm_export_base::get_peers()

Returns the list of all bindings of the export.

### SystemVerilog

```
function void get_peers(vmm_tlm_port_base#(DATA,PHASE)
peers[$]);
```

### Description

Returns the queue of bindings of the export in the specified queue.

### Example

```
class consumer extends vmm_xactor;
   vmm_tlm_b_transport_export#(consumer) b_export;
   function display_connections();
      vmm_tlm_port_base q[$];
      b_export.get_peers(q);
      foreach(q[i])
         $display("Binding[%0d] %s",i,
            q[i].get_object_name() );
   endfunction
endclass
```

## vmm_tlm_export_base::new()

Constructor of an export base class.

### SystemVerilog

```
function new(vmm_object parent, string name,
   int max_binds = 1, int min_binds = 0, vmm_log log);
```

### Description

Sets the parent, if it is an extension of `vmm_object`. Sets the name of the instance. Sets the maximum and minimum bindings allowed for this export. `log` is the message interface instance to be used for reporting messages.

## vmm_tlm_export_base::tlm_bind()

Binds the TLM export to the TLM port passed as an argument.

### SystemVerilog

```
function void tlm_bind(vmm_tlm_port_base#(DATA,PHASE)
   peer,int id = -1, string fname = "", int lineno = 0);
```

### Description

Binds the TLM export to the supplied port. Multiple bindings are allowed for exports.

This method adds the supplied port descriptor to the bindings list of the export. An error is generated, if the supplied port already contains a binding.

The second argument, `id`, is used to distinguish between multiple ports that bind to the same export. If a positive `id` is supplied, then it must be unique for this export. It is an error, if a positive `id` already used by the export is supplied. If no `id` or a negative `id` is provided, then the lowest available positive `id` is automatically assigned. This `id` is passed as an argument of the transport method, implemented in the exports parent.

The `fname` and `lineno` arguments are used to track the file name and the line number, where the `tlm_bind` is invoked from.

### Example

```
class producer extends vmm_xactor;
   vmm_tlm_b_transport_port#(producer)
      b_port = new(this,"producer port");
endclass
```

```
class consumer extends vmm_xactor;
   vmm_tlm_b_transport_export#(consumer)
      b_export = new(this,"consumer export");
   function b_transport(int index=-1, vmm_data trans,
                        ref int delay);
      if(index == 0 )
         $display("From producer 0");
      else if (index == 1)
      ...
   endfunction
endclass

class my_env extends vmm_group;
   producer p[4];
   consumer c;

   function void connect_ph();
      foreach(p[i]) begin
         c.b_export.tlm_bind(p[i].b_port, i);
      end
   endfunction
endclass
```

## vmm_tlm_export_base::tlm_import()

Imports an export from an inner level in the hierarchy, to an outer level.

### SystemVerilog

```
function void tlm_import(vmm_tlm_export_base#(DATA,PHASE)
   peer, string fname = "", int lineno = 0);
```

### Description

This is a special way of exporting bindings. It simplifies the binding for hierarchical exports, by making the inner export visible to the outer hierarchy. The binding resolves to a port-export binding. The method allows only parent-child exports to be imported. An error is generated, if the exports do not share a parent-child relationship. It is an error to import an export that is already imported. It is an error to import an export that is already bound. The method can be called for both parent-to-child bindings and child-to-parent bindings. For this, the parent transactors must be derivatives of `vmm_object`. If the parent is a `vmm_xactor` extension, then the `vmm_xactor` base class should be underpinned. If the `vmm_xactor` is not underpinned, or the parent is not a derivative of `vmm_object`, then only `child.export.tlm_import(parent.export)` is allowed. The error checks are not executed, and you must ensure legal connections.

The `fname` and `lineno` arguments are used to track the file name and the line number, where the `tlm_import` is invoked from.

### Example

```
class target_child extends vmm_xactor;
```

```
   vmm_tlm_b_transport_export#(target_child) b_export;
endclass

class target_parent extends vmm_group;
   vmm_tlm_b_transport_export#(target_parent) b_export;
   target_child target;
   function void connect_ph();
      target.b_export.tlm_import(this.b_export);
   endfunction
endclass
```

## vmm_tlm_export_base::tlm_unbind()

Removes an existing binding of the export.

### SystemVerilog

```
function void tlm_unbind(vmm_tlm_port_base#(D,P)
   peer = null, int id = -1, string fname = "", int lineno = 0);
```

### Description

Removes the binding supplied as a *`peer`* or `id` from the list of bindings, for this export. Also, removes the binding of this export with the connected port.

If the supplied *`peer`* is not null, then the binding of the *`peer`* is removed. An error is generated, if the supplied *`peer`* is not bound to this export.

If the supplied *`peer`* is null and the supplied `id` is a positive number, then the binding to the port with the supplied `id` is removed. An error is generated, if there is no binding with the supplied positive `id`.

If the supplied *`peer`* is null and the supplied `id` negative, then all bindings for this export are removed.

On unbinding, the `id` of the unbound port becomes available for reuse.

The `fname` and `lineno` arguments are used to track the file name and the line number, where the `tlm_unbind` is invoked from.

## Example

```
class my_env extends vmm_group;
   producer p1;
   consumer c1, c2;
   function void connect_ph()
      p1.b_port.tlm_bind(c1.b_export);
   endfunction

class test2 extends vmm_test;
   function void configure_test_ph();
      env.p1.b_port.tlm_unbind();
      env.p1.b_port.tlm_bind(c2.b_export);
   endfunction
endclass
```

# vmm_tlm_nb_transport_bw_export#(T,D,P)

Non-blocking backward transport export class.

## SystemVerilog

```
class vmm_tlm_nb_transport_bw_export#(
   type TARGET = vmm_tlm_xactor, type DATA = vmm_data,
   type PHASE = vmm_tlm::phase_e)
   extends vmm_tlm_export_base#(DATA,PHASE);
```

## Description

Class providing the non-blocking backward transport export. This class should be instantiated in the initiator transactor, which instantiates a non-blocking forward port. The transactions sent from this transactor, on the forward path, can be received by the transactor on the backward path through this backward export.

The parameter type `TARGET` is the class instantiating the transport export. This defaults to `vmm_tlm_xactor`. The parameter `DATA` is the data type of the transaction in the export services. The default is `vmm_data`. The parameter type, `PHASE`, is the phase class for this export. The default type is `vmm_tlm::phase_e`.

The export can be bound to multiple ports, up to the max bindings specified in the constructor of this class.

## Summary


- `vmm_tlm_nb_transport_bw_export() ..... page B-326
- vmm_tlm_nb_transport_bw_export::nb_transport_bw() page B-328
- vmm_tlm_nb_transport_bw_export::new() ..... page B-329

## `vmm_tlm_nb_transport_bw_export()

Shorthand macro to create unique instances of non-blocking, backward transport export. This is useful if multiple exports are required in the same initiator transactor.

### SystemVerilog

```
`vmm_tlm_nb_transport_bw_export(SUFFIX)
```

### Description

This macro creates a uniquified `vmm_tlm_nb_transport_bw_export` class, with the *`SUFFIX`* appended to the class name `vmm_tlm_nb_transport_bw_export`. The class with the name `vmm_tlm_nb_transport_bw_export`*`SUFFIX`* is created in the scope, where the macro is called.

This macro is required if there are multiple instances of the `vmm_tlm_nb_transport_bw_export` class, and each requires a unique implementation of the `nb_transport_bw()` task, in the parent transactor.

The `nb_transport_bw()` methods in the parent transactor must be uniquified, using the same *`SUFFIX`* to `nb_transport_bw`.

Alternatively, if multiple ports need to service the parent transactor, then a single export with multiple bindings using unique ids can be used in place of the macro. The single `nb_transport_bw()` method can be programmed to serve various ports, depending on the id.

## Example

```
class producer extends vmm_xactor;
   `vmm_tlm_nb_transport_bw_export(_1)
   `vmm_tlm_nb_transport_bw_export(_2)
   vmm_tlm_nb_transport__bw_export_1#(producer)
      nb_export1 = new(this, "export1");
   vmm_tlm_nb_transport_bw_export_2#(producer)
      nb_export2 = new(this, "export2");
   function nb_transport_bw_1(int id = -1, vmm_data trans,
         ref vmm_tlm::phase_e ph, ref int delay);
      trans.display("From export1");
   endfunction
   function nb_transport_bw_2(int id = -1,vmm_data trans,
         ref vmm_tlm::phase_e ph, ref int delay);
      trans.display("From export2");
   endfunction
endclass

class consumer extends vmm_xactor;
   vmm_tlm_nb_transport_bw_port#(producer) nb_port;
endclass

class my_env extends vmm_group;
   producer p1;
   consumer c1,c2;
   function void connect_ph();
      p1.nb_export1.tlm_bind(c1.nb_port);
      p1.nb_export2.tlm_bind(c2.nb_port);
   endfunction
endclass
```

## vmm_tlm_nb_transport_bw_export::nb_transport_bw()

Non-blocking transport method of the export.

### SystemVerilog

```
function vmm_tlm::sync_e nb_transport_bw(int id,
   DATA trans, ref PHASE ph, ref int delay );
```

### Description

Non-blocking transport function of the transport export. This function is internally called by the bound transport port. This function calls the `nb_transport_bw()` method of parent transactor it is instantiated in.The argument `id` specifies the binding id of this export. If the export is bound to multiple ports then the peer can be distinguished using the `id` passed to the `nb_transport_bw`().The `trans` argument is a handle of the transaction object, `ph` is the handle of phase class to specify the phase of a transaction `trans`, the `delay` argument is the timing annotation.

### Example

```
class producer extends vmm_xactor;
   vmm_tlm_nb_transport_fw_export#(producer) nb_export;
   function vmm_tlm::sync_e nb_transport_bw(int id = -1,
      vmm_data trans,ref vmm_tlm::phase_e ph, ref  int delay
);
      trans.display("From producer on backward path.");
   endfunction
endclass
```

## vmm_tlm_nb_transport_bw_export::new()

Constructor of non-blocking backward transport export class. Any class instantiating this non-blocking export must provide an implementation of the `nb_transport_bw()` function.

### SystemVerilog

```
function new(TARGET parent, string name, int max_binds = 1 ,
   int min_binds = 0);
```

### Description

Sets the parent and instance name of the blocking transport export. Sets the maximum and minimum bindings allowed for this export. The default value of maximum bindings is 1 and minimum bindings is 0. An error is generated during `tlm_bind()`, if the current binding exceeds the maximum allowed bindings for the export. An error is generated during elaboration, if the export does not contain the minimum number of specified bindings.

### Example

```
class producer extends vmm_xactor;
   vmm_tlm_nb_transport_bw_export#(producer) nb_export;
   vmm_tlm_nb_transport_fw_port#(producer)nb_port;
   function void build_ph();
      this.nb_export = new(this,"consumer export",5,1);
   endfunction
endclass
```

# vmm_tlm_nb_transport_bw_port#(I,D,P)

Non-blocking transport port for the backward path.

## SystemVerilog

```
class vmm_tlm_nb_transport_bw_port #(
   type INITIATOR = vmm_tlm_xactor, type DATA = vmm_data,
   type PHASE = vmm_tlm::phase_e)
   extends vmm_tlm_port_base#(DATA,PHASE);
```

## Description

Class providing the non-blocking backward transport port. Transactions received from the producer, on the forward path, are sent back to the producer on the backward path using this non-blocking transport port. The parameter type `INITIATOR` is the class instantiating the transport port. This defaults to `vmm_tlm_xactor`. The parameter `DATA` is the data type of the transaction the port services. The default is `vmm_data`. The parameter type `PHASE` is the phase class for this port. The default type is `vmm_tlm::phase_e`.

The port can be bound to one export. A warning is generated if the port is left unbound.

## Summary


- vmm_tlm_nb_transport_bw_port::nb_transport_bw() .. page B-331
- vmm_tlm_nb_transport_bw_port::new() ..... page B-332

## vmm_tlm_nb_transport_bw_port::nb_transport_bw()

Non-blocking backward transport function. The target transactor instantiating this transport port should call the `nb_transport_bw()` method of the transport port.

### SystemVerilog

```
function vmm_tlm::sync_e nb_transport_bw(DATA trans,
   ref PHASE ph, ref int delay );
```

### Description

Non-blocking transport function of the port. Calls the `nb_transport_bw()` method of the bound export. The argument `trans` is a handle of the transaction object, `ph` is a handle of the phase class, and `delay` is the timing annotation.

### Example

```
class consumer extends vmm_xactor;
   vmm_tlm_nb_transport_bw_port#(consumer) nb_port;
   my_trans current_trans ;
   task run_ph();
      while(1) begin
         this.nb_port.nb_tranport_bw(current_trans,ph,
         delay);
         #5;
      end
   endtask
endclass
```

## vmm_tlm_nb_transport_bw_port::new()

Constructor of non-blocking backward transport port class.

### SystemVerilog

```
function new(INITIATOR parent, string name);
```

### Description

Sets the parent and instance name of the non-blocking backward transport port.

### Example

```
class consumer extends vmm_xactor;
   vmm_tlm_nb_transport_fw_export#(consumer) nb_export;
   vmm_tlm_nb_transport_bw_port#(consumer)nb_port ;
   function void build_ph();
      this.nb_port = new(this,"consumer port");
   endfunction
endclass
```

# vmm_tlm_nb_transport_export#(T,D,P)

Bidirectional non-blocking export.

## SystemVerilog

```
class vmm_tlm_nb_transport_export#(
   type TARGET = vmm_tlm_xactor, type DATA = vmm_data,
   type FW_PHASE = vmm_tlm, type BW_PHASE = FW_PHASE)
   extends vmm_tlm_socket_base#(DATA,BW_PHASE);
```

## Description

Bidirectional export providing non-blocking transport export for the forward path, and non-blocking transport port for the backward path in a single transport export.

Only one-to-one binding is allowed for this bidirectional non-blocking export. The `vmm_tlm_nb_transport_export` can only be bound to the `vmm_tlm_nb_transport_port`.

The `vmm_tlm_socket_base` provides all the access methods that are provided by the `vmm_tlm_export_base` class. The methods available with this class are `tlm_bind()`, `tlm_unbind()`, `tlm_import()`, and `get_peer()`. For more information on the descriptions of those methods, see the `vmm_tlm_port_exbase` class description.

This class provides non-blocking transport methods for both the forward path, `nb_transport_fw` and the backward path, `nb_transport_bw`.

Any transactor class instantiating this bidirectional export must provide an implementation of the `nb_transport_fw()` method, and should call the `nb_transport_bw()` method of this export.

## Example

```
class consumer extends vmm_xactor;
   vmm_tlm_nb_transport_export#(consumer) nb_export =
      new(this,"consumer_bi");

   function vmm_tlm::sync_e nb_transport_fw(int id=-1,
      my_trans trans, ref vmm_tlm ph, ref int delay);
   endfunction

   virtual task run_ph();
      my_trans tr;
      while(1) begin
         this.tr.notify.wait_for(vmm_data::ENDED);
         this.nb_port.nb_transport_bw(tr,ph,delay);
         #5;
      end
   endtask
endclass
```

## Summary


- `vmm_tlm_nb_transport_export() ..... page B-335

## `vmm_tlm_nb_transport_export()

Shorthand macro to create unique classes of the bidirectional export. This is useful if multiple `vmm_tlm_nb_transport_export` instances are required in the same initiator transactor, each having its own implementation of the `nb_transport_fw()` method.

### SystemVerilog

```
`vmm_tlm_nb_transport_export(SUFFIX)
```

### Description

The use model is similar to the shorthand macros provided for the unidirectional non-blocking exports. For more information, see the description of `` `vmm_tlm_nb_transport_fw_export `` macro.

### Example

```
class consumer extends vmm_xactor;
   `vmm_tlm_nb_transport_export(_1)
   `vmm_tlm_nb_transport_export(_2)
   vmm_tlm_nb_transport_export_1#(producer) nb_exp1;
   vmm_tlm_nb_transport_export_2#(producer) nb_exp2;

   function vmm_tlm::sync_e nb_transport_fw_1(int id=-1,
      my_trans trans,ref vmm_tlm ph, ref int delay);
   endfunction

   function vmm_tlm::sync_e nb_transport_fw_2(int id=-1,
      my_trans trans,ref vmm_tlm ph, ref int delay);
   endfunction
endclass
```

# vmm_tlm_nb_transport_fw_export#(T,D,P)

Non-blocking forward transport export class.

## SystemVerilog

```
class vmm_tlm_nb_transport_fw_export#(
   type TARGET = vmm_tlm_xactor, type DATA = vmm_data,
   type PHASE = vmm_tlm::phase_e)
   extends vmm_tlm_export_base#(DATA,PHASE);
```

## Description

Class providing the non-blocking forward transport export. The parameter type `TARGET` is the class instantiating the transport export. This defaults to `vmm_tlm_xactor`. The parameter `DATA` is the data type of the transaction the export services. The default is `vmm_data`. The parameter type `PHASE` is the phase class for this export. The default type is `vmm_tlm::phase_e`.

The export can be bound to multiple ports up to the max bindings specified in the constructor of this class.

## Summary


- `vmm_tlm_nb_transport_fw_export() ..... page B-337
- vmm_tlm_nb_transport_fw_export::nb_transport_fw() page B-339
- vmm_tlm_nb_transport_fw_export::new() ..... page B-340

## `vmm_tlm_nb_transport_fw_export()

Shorthand macro to create unique instances of non-blocking forward transport export. This is useful if multiple exports are required in the same target transactor.

### SystemVerilog

```
`vmm_tlm_nb_transport_fw_export(SUFFIX)
```

### Description

This macro creates a uniquified `vmm_tlm_nb_transport_fw_export` class, with *`SUFFIX`* appended to the `vmm_tlm_nb_transport_fw_export` class name. The class with the name `vmm_tlm_nb_transport_fw_export`*`SUFFIX`* is created in the scope, where the macro is called.

This macro is required if there are multiple instances of the `vmm_tlm_nb_transport_fw_export` class, and each requires a unique implementation of the `nb_transport_fw()` task in the parent transactor.

The `nb_transport_fw()` methods in the parent transactor must be uniquified using the same *`SUFFIX`* to `nb_transport_fw`.

Alternatively, if multiple ports need to service the parent transactor, then a single export with multiple bindings using unique ids can be used in place of the macro. The single `nb_transport_fw()` method can be programmed to serve the various ports depending on the id.

## Example

```
class consumer extends vmm_xactor;
   `vmm_tlm_nb_transport_fw_export(_1)
   `vmm_tlm_nb_transport_fw_export(_2)
   vmm_tlm_nb_transport__fw_export_1#(consumer)
      nb_export1 = new(this, "export1");
   vmm_tlm_nb_transport_fw_export_2#(consumer)
      nb_export2 = new(this, "export2");

   function nb_transport_fw_1(int id = -1, vmm_data trans,
         ref vmm_tlm::phase_e ph, ref int delay );
      trans.display("From export1");
   endfunction

   task nb_transport_fw_2(int id = -1, vmm_data trans,
         ref vmm_tlm::phase_e ph, ref int delay);
      trans.display("From export2");
   endtask
endclass

class producer extends vmm_xactor;
   vmm_tlm_nb_transport_fw_port#(producer) nb_port;
endclass

class my_env extends vmm_group;
   producer p1,p2;
   consumer c1;

   function void connect_ph();
      c1.nb_export1.tlm_bind(p1.nb_port);
      c1.nb_export2.tlm_bind(p2.nb_port);
   endfunction
endclass
```

## vmm_tlm_nb_transport_fw_export::nb_transport_fw()

Non-blocking transport method of the export.

### SystemVerilog

```
function vmm_tlm::sync_e nb_transport_fw(int id = -1,
   DATA trans, ref PHASE ph, ref int delay);
```

### Description

Non-blocking transport function of the transport export. This function is internally called by the bound transport port. This function calls the `nb_transport_fw()` method of the parent transactor in which it is instantiated. If the export is bound to multiple ports then the peer can be distinguished using the `id` field passed to the `nb_transport_fw()` method.

The `trans` argument is a handle of the transaction object, `ph` is the handle of phase class to specify the phase of a transaction `trans`, and the `delay` argument is the timing annotation.

### Example

```
class consumer extends vmm_xactor;
   vmm_tlm_nb_transport_fw_export#(consumer) nb_export;
   function vmm_tlm::sync_e nb_transport_fw(int id = -1,
      vmm_data trans,ref vmm_tlm::phase_e ph,
      ref int delay );
      trans.display("From consumer");
      return vmm_tlm::TLM_COMPLETED;
   endfunction
endclass
```

## vmm_tlm_nb_transport_fw_export::new()

Constructor of non-blocking forward transport export class. Any class instantiating this non-blocking export must provide an implementation of the `nb_transport_fw()` function.

### SystemVerilog

```
function new(TARGET parent, string name, int max_binds = 1,
   int min_binds = 0);
```

### Description

Set the parent and instance name of the blocking transport export. Sets the maximum and minimum bindings allowed for this export. The default value of maximum bindings is 1 and minimum binding is 0. An error is issued during `tlm_bind()` if the current binding exceeds the maximum allowed bindings for the export. An error is issued during elaboration if the export does not have the minimum number of specified bindings.

### Example

```
class consumer extends vmm_xactor;
   vmm_tlm_nb_transport_fw_export#(consumer) nb_export;
   function void build_ph();
      this.nb_export = new(this,"consumer export",5,1);
   endfunction
endclass
```

# vmm_tlm_nb_transport_fw_port#(I,D,P)

Non-blocking transport port for the forward path.

## SystemVerilog

```
class vmm_tlm_nb_transport_fw_port #(
   type INITIATOR=vmm_tlm_xactor,
   type DATA = vmm_data, type PHASE = vmm_tlm::phase_e)
   extends vmm_tlm_port_base#(DATA,PHASE);
```

## Description

Class providing the non-blocking forward transport port. Transactions originating from the producer are sent on the forward path, using this non-blocking transport port. The parameter type, `INITIATOR` is the class that instantiates the transport port. This defaults to `vmm_tlm_xactor`. The parameter `DATA` is the data type of the transaction the port services. The default is `vmm_data`. The parameter type `PHASE` is the phase class for this port. The default type is `vmm_tlm::phase_e`.

The port can be bound to one export. A warning is generated if the port is left unbound.

## Summary


- vmm_tlm_nb_transport_fw_port::nb_transport_fw() .. page B-342
- vmm_tlm_nb_transport_fw_port::new() ..... page B-343

## vmm_tlm_nb_transport_fw_port::nb_transport_fw()

Non-blocking forward transport function. The initiator transactor initiating this transport port should call the `nb_transport_fw()` method of the transport port.

### SystemVerilog

```
function vmm_tlm::sync_e nb_transport_fw(DATA trans,
   ref PHASE ph, ref int delay);
```

### Description

Call the `nb_transport_fw()` method of the bound export. The argument, `trans` is a handle of the transaction object, `ph` is a handle of the phase class, and `delay` is the timing annotation.

You must ensure that delay is provided in the loop, where this non-blocking function is being called.

### Example

```
class producer extends vmm_xactor;
   vmm_tlm_nb_transport_port#(producer) nb_port;
   task run_ph();
      my_data tr;
      while(1) begin
         tr = new();
         this.nb_port.nb_tranport_fw(tr,ph,delay);
         #5;
      end
   endtask
endclass
```

## vmm_tlm_nb_transport_fw_port::new()

Constructor of non-blocking forward transport port class.

### SystemVerilog

```
function new(TARGET parent, string name);
```

### Description

Sets the parent and instance name of the non-blocking forward transport port.

### Example

```
class producer extends vmm_xactor;
   vmm_tlm_nb_transport_fw_port#(producer) nb_port;
   function void build_ph();
      this.nb_port = new(this,"producer port");
   endfunction
endclass
```

# vmm_tlm_nb_transport_port#(I,D,P)

Bidirectional non-blocking port.

## SystemVerilog

```
class vmm_tlm_nb_transport_port#(
   type INITIATOR = vmm_tlm_xactor, type DATA = vmm_data,
   type FW_PHASE = vmm_tlm::phase_e,
   type BW_PHASE = FW_PHASE)
   extends vmm_tlm_socket_base#(DATA,FW_PHASE);
```

## Description

Bidirectional port providing a non-blocking transport port for the forward path, and a non-blocking transport export for the backward path, in a single transport port.

Only one-to-one binding is allowed for this bidirectional, non-blocking port. The `vmm_tlm_nb_transport_port` can only be bound to the `vmm_tlm_nb_transport_export`.

The `vmm_tlm_socket_base` provides all the access methods that are provided by the `vmm_tlm_port_base` class. The methods available with this class are `tlm_bind()`, `tlm_unbind()`, `tlm_import()`, and `get_peer()`. For more information on those methods, see the `vmm_tlm_port_base` class.

This class provides non-blocking transport methods for both, the forward path, `nb_transport_fw` and the backward path, `nb_transport_bw`.

Any transactor class instantiating this bidirectional port must provide an implementation of the `nb_transport_bw()` method, and should call the `nb_transport_fw()` method of this port.

## Example

```
class producer extends vmm_xactor;
   vmm_tlm_nb_transport_port#(producer) nb_port =
      new(this,"producer_bi");

   function vmm_tlm::sync_e nb_transport_bw(int id=-1,
      my_trans trans, ref vmm_tlm::phase_e ph,
      ref int delay);
   endfunction

   virtual task run_ph();
      my_trans tr;
      while(1) begin
         tr = new();
         tr.randomize();
         this.nb_port.nb_transport_fw(tr,ph,delay);
         #5;
      end
   endtask
endclass
```

## Summary


- `vmm_tlm_nb_transport_port() ..... page B-346

## `vmm_tlm_nb_transport_port()

Shorthand macro to create unique classes of the bidirectional port. This is useful if multiple `vmm_tlm_nb_transport_port` instances are required in the same initiator transactor, each having its own implementation of the `nb_transport_bw()` method.

### SystemVerilog

```
`vmm_tlm_nb_transport_port(SUFFIX)
```

### Description

The use model is similar to the shorthand macros provided for the unidirectional non-blocking ports. For more information, see the description of the `` `vmm_tlm_nb_transport_fw_export `` macro.

### Example

```
class producer extends vmm_xactor;
   `vmm_tlm_nb_transport_port(_1)
   `vmm_tlm_nb_transport_port(_2)
   vmm_tlm_nb_transport_port_1#(producer) nb_port1;
   vmm_tlm_nb_transport_port_2#(producer)
 nb_port2;
   function vmm_tlm::sync_e nb_transport_bw_1
      (int id=-1,my_trans trans,ref vmm_tlm::phase_e ph,
      ref int delay);
   endfunction

   function vmm_tlm::sync_e nb_transport_bw_2
      (int id=-1,my_trans trans,ref vmm_tlm::phase_e ph,
      ref int delay);
   endfunction
endclass
```

# vmm_tlm_port_base#(D,P)

Abstract base class for all TLM2.0 transport ports

## SystemVerilog

```
virtual class vmm_tlm_port_base#(type DATA=vmm_data,
  type PHASE = vmm_tlm::phase_e) extends vmm_tlm_base;
```

## Description

This is an abstract base class for all TLM2.0 transport ports. This class contain the methods that are required by all TLM2.0 transport port implementations. Any user-defined port must be extended from this base class.

The `DATA` parameter is the type of the transaction object the port services. The default type is `vmm_data`. The `PHASE` parameter is the type of the phasing class. The default value is `vmm_tlm::phase_e`.

## Summary


- vmm_tlm_port_base::get_peer() ..... page B-348
- vmm_tlm_port_base::get_peer_id() ..... page B-349
- vmm_tlm_port_base::new() ..... page B-350
- vmm_tlm_port_base::tlm_bind() ..... page B-351
- vmm_tlm_port_base::tlm_import() ..... page B-353
- vmm_tlm_port_base::tlm_unbind() ..... page B-355

## vmm_tlm_port_base::get_peer()

Returns the binding for the port.

### SystemVerilog

```
function vmm_tlm_export_base#(DATA,PHASE) get_peer();
```

### Description

Returns the export bound to the current port. Returns Null, if the port does not contain a binding.

### Example

```
class producer extends vmm_xactor;
   vmm_tlm_b_transport_port#(producer) b_port;

   function display_my_id();
      vmm_tlm_export_base peer;
      peer = this.b_port.get_peer();
      $display("My id = %d",peer.get_peer_id(this);
   endfunction
endclass
```

## vmm_tlm_port_base::get_peer_id()

Returns the `id` of this port for its binding

### SystemVerilog

```
function int get_peer_id();
```

### Description

Returns the `id` of this port, with respect to its export binding. If port is not bound, -1 is returned.

### Example

```
class my_env extends vmm_group;
   producer p1,p2;
   consumer c1;

   function connect_ph();
      p1.b_port.tlm_bind(c1.b_export);
      p2.b_port.tlm_bind(c1.b_export);
      int p1_id = p1.b_port.get_peer_id(); //returns 0
   int p2_id = p2.b_port.get_peer_id(); //returns 1
   endfunction
endclass
```

## vmm_tlm_port_base::new()

Constructor of the port base class.

### SystemVerilog

```
function new(vmm_object parent, string name, vmm_log log);
```

### Description

Sets the parent, if the base class extends `vmm_object`. Sets the name of the instance. `log` is the message interface instance to be used for reporting messages.

## vmm_tlm_port_base::tlm_bind()

Binds the TLM port to the TLM export passed as an argument.

### SystemVerilog

```
function void tlm_bind(vmm_tlm_export_base#(DATA,PHASE)
   peer,int id = -1, , string fname = "", int lineno = 0);
```

### Description

Binds the TLM port to the TLM export. A port can contain only one binding, though multiple bindings are allowed for exports. It is an error to bind a port that already contains a binding.

This method adds the current port descriptor to the bindings list of `peer`. Calling `port.tlm_bind(export,id)` is equivalent to `export.tlm_bind(port,id)`, and the binding can be done either way. It is an error if both calls are made, since the port allows only one binding.

The second argument, `id`, is used to distinguish between multiple ports that bind to the same export. The `id` field is used by the export. If a positive `id` is supplied, then it must be unique for that export. It is an error if a positive `id` already used by the export is supplied. If no `id` or a negative `id` is supplied, then the lowest available unique positive `id` is automatically assigned. This `id` is passed as an argument of the transport method implemented in the exports parent. The `fname` and `lineno` arguments are used to track the file name and the line number, where `tlm_bind` is invoked from.

### Example

```
class producer extends vmm_xactor;
```

```
   vmm_tlm_b_transport_port#(producer)
      b_port = new(this,"producer port");
endclass

class consumer extends vmm_xactor;
   vmm_tlm_b_transport_export#(consumer)
      b_export = new(this,"consumer export");
endclass

class my_env extends vmm_group;
   producer p[4];
   consumer c;

   function void connect_ph();
      foreach(p[i]) begin
         p[i].b_port.tlm_bind(c.b_export, i);
      end
   endfunction
endclass
```

## vmm_tlm_port_base::tlm_import()

Imports a port from an inner level in the hierarchy, to an outer level.

### SystemVerilog

```
function void tlm_import(vmm_tlm_port_base#(DATA,PHASE)
peer, string fname = "", int lineno = 0);
```

### Description

This is a special port-to-port binding. It simplifies the binding for hierarchical ports and exports, by making the inner port visible to the outer hierarchy. The binding finally resolves to a port-export binding.

The method allows only parent-child ports to be imported. An error is generated, if the ports do not share a parent-child relationship. It is an error to import a port that is already imported. It is an error to import a port that is already bound.

The method can be called for both parent-to-child binding and child-to-parent binding. The parent transactors must be derivatives of `vmm_object`. If the parent is a `vmm_xactor` extension, then the `vmm_xactor` base class should be underpinned. If the `vmm_xactor` is not underpinned, or the parent is not a derivative of `vmm_object`, then only *`child`*`.port.tlm_import(`*`parent`*`.port)` is allowed. The error checks are not executed, and you must ensure legal connections. The `fname` and `lineno` arguments are used to track the file name and the line number, where `tlm_import` is invoked from.

### Example

```
class initiator_child extends vmm_xactor;
```

```
   vmm_tlm_b_transport_port#(initiator_child) b_port;
endclass

class initiator_parent extends vmm_group;
   vmm_tlm_b_transport_port#(initiator_parent) b_port;
   initiator_child initiator;
   function void connect_ph();
      initiator.b_port.tlm_import(this.b_port);
   endfunction
endclass

class target extends vmm_xactor;
   vmm_tlm_b_transport_export b_export;
endclass

class my_env extends vmm_group;
   initiator_parent initiator;
   target target;
   function void connect_ph();
      initiator.b_port.tlm_bind(target.b_export);
   endfunction
endclass
```

## vmm_tlm_port_base::tlm_unbind()

Removes the existing port binding.

### SystemVerilog

```
function void tlm_unbind(string fname = "", int lineno = 0);
```

### Description

Sets the port binding to null. Also, removes the current port descriptors binding from the export that the port is bound to. A warning is generated if a binding does not exist for this port.

This method can be used to dynamically change existing bindings for a port. The `fname` and `lineno` arguments are used to track the file name and the line number, where `tlm_unbind` is invoked from.

### Example

```
class my_env extends vmm_group;
   producer p1;
   consumer c1, c2;
   function void connect_ph();
      p1.b_port.tlm_bind(c1.b_export);
   endfunction
endclass
class test2 extends vmm_test;
   function void configure_test_ph();
      env.p1.b_port.tlm_unbind();
      env.p1.b_port.tlm_bind(c2.b_export);
   endfunction
endclass
```

# vmm_tlm_initiator_socket#(I,D,P)

Bidirectional socket port providing both blocking and non-blocking paths.

## SystemVerilog

```
class vmm_tlm_initiator_socket#(
   type INITIATOR = vmm_tlm_xactor, type DATA = vmm_data,
   type PHASE = vmm_tlm::phase_e)
   extends vmm_tlm_socket_base#(DATA,PHASE);
```

## Description

Bidirectional socket port providing blocking transport port, non-blocking transport port for the forward path, and non-blocking transport export for the backward path, in a single transport socket.

Only one-to-one binding is allowed for this bidirectional socket. The `vmm_tlm_initiator_socket` can only be bound to the `vmm_tlm_target_socket`.

The `vmm_tlm_socket_base` provides all access methods that are provided by the `vmm_tlm_port_base` class. The methods available with this class are `tlm_bind()`, `tlm_unbind()`, `tlm_import()`, and `get_peer()`. For more information on those methods, see the `vmm_tlm_port_base` class description.

This class provides a blocking `b_transport()` transport method, and non-blocking `nb_transport_fw()` and `nb_transport_bw()` transport methods for the forward path and the backward path, respectively.

Any transactor class instantiating this bidirectional socket must provide an implementation of the `nb_transport_bw()` method, and should call one or both of the `b_transport()` and `nb_transport_fw()` methods of this socket.

## Example

```
class producer extends vmm_xactor;
   vmm_tlm_initiator_socket#(producer) socket =
      new(this,"producer_socket");

   function vmm_tlm::sync_e nb_transport_bw(int id=-1,
      my_trans trans, ref vmm_tlm ph, ref int delay);
   endfunction

   virtual task run_ph();
      my_trans tr;
      while(1) begin
         tr = new();
         tr.randomize();
         this.socket.nb_transport_fw(tr,ph,delay);
         #5;
      end
   endtask
endclass
```

## Summary


- `vmm_tlm_initiator_socket() ..... page B-358

## `vmm_tlm_initiator_socket()

Shorthand macro to create unique classes of the bidirectional socket. This is useful if multiple `vmm_tlm_initiator_socket` instances are required in the same initiator transactor, each having its own implementation of the `nb_transport_bw()` method.

### SystemVerilog

```
`vmm_tlm_initiator_socket(SUFFIX)
```

### Description

The use model is similar to the shorthand macros, provided for the unidirectional non-blocking ports. For more information, see the description of the `` `vmm_tlm_nb_transport_fw_export `` macro.

### Example

```
class producer extends vmm_xactor;
   `vmm_tlm_initiator_socket(_1)
   `vmm_tlm_initiator_socket(_2)
   vmm_tlm_initiator_socket_1#(producer) s1;
   vmm_tlm_initiator_socket_2#(producer) s2;

   function vmm_tlm::sync_e nb_transport_bw_1(
      int id=-1,my_trans trans,ref vmm_tlm::phase ph,
      ref int delay);
   endfunction

   function vmm_tlm::sync_e nb_transport_bw_2(
      int id=-1,my_trans trans,ref vmm_tlm::phase_e ph,
      ref int delay);
   endfunction
endclass
```

# vmm_tlm_target_socket#(T,D,P)

Bidirectional socket export providing both blocking and non-blocking paths.

## SystemVerilog

```
class vmm_tlm_target_socket#(
   type TARGET = vmm_tlm_xactor, type DATA = vmm_data,
   type PHASE = vmm_tlm::phase_e)
   extends vmm_tlm_socket_base#(DATA,PHASE);
```

## Description

Bidirectional socket export providing blocking transport export, non-blocking transport export for the forward path, and non-blocking transport port for the backward path in a single transport socket.

Only one-to-one binding is allowed for this bidirectional socket. The `vmm_tlm_target_socket` can only be bound to the `vmm_tlm_initiator_socket`.

The `vmm_tlm_socket_base` provides all access methods that are provided by the `vmm_tlm_port_base` class. The methods available with this class are `tlm_bind()`, `tlm_unbind()`, `tlm_import()`, and `get_peer()`. For more information on these methods, refer to the `vmm_tlm_port_base` class description.

This class provides a blocking `b_transport()` transport method, and non-blocking `nb_transport_fw()` and `nb_transport_bw`() transport methods for the forward and backward paths, respectively.

Any transactor class instantiating this bidirectional socket must provide an implementation of the `nb_transport_fw()` and `b_transport()` methods, and should call `nb_transport_bw()` method of this socket for the backward path.

## Example

```
class consumer extends vmm_xactor;
   vmm_tlm_target_socket#(consumer) nb_export =
         new(this,"consumer_socket");
   function vmm_tlm::sync_e nb_transport_fw(int id=-1,
         my_trans trans,ref vmm_tlm::phase_e ph,
         ref int delay);
   endfunction

   task b_transport(int id=-1, my_trans trans,
                    ref int delay);
   endtask

   virtual task run_ph();
      my_trans tr;
      while(1) begin
         this.tr.notify.wait_for(vmm_data::ENDED);
         this.nb_port.nb_transport_bw(tr,ph,delay);
         #5;
      end
   endtask
endclass
```

## Summary


- `vmm_tlm_target_socket() ..... page B-361

## `vmm_tlm_target_socket()

Shorthand macro to create unique classes of the bidirectional socket. Used if multiple `vmm_tlm_target_socket` instances are required in the same target transactor, each having its own implementation of the `nb_transport_bw()` method.

### SystemVerilog

```
`vmm_tlm_nb_simple_target_socket(SUFFIX)
```

### Description

The use model is similar to the shorthand macros provided for the unidirectional exports. For more information, see the `` `vmm_tlm_nb_transport_fw_export `` macro description.

### Example

```
class consumer extends vmm_xactor;
   `vmm_tlm_target_socket(_1)
   `vmm_tlm_target_socket(_2)
   vmm_tlm_target_socket_1#(producer) soc1;
   vmm_tlm_target_socket_2#(producer) soc2;

   function vmm_tlm::sync_e nb_transport_fw_1(
      int id=-1,my_trans trans, ref vmm_tlm::phase_e ph,
      ref int delay);
   endfunction

   function vmm_tlm::sync_e nb_transport_fw_2(
      int id=-1,my_trans trans, ref vmm_tlm::phase_e ph,
      ref int delay);
   endfunction
   task b_transport_1(int id=-1, my_trans trans,
                      ref int delay);
   endtask
```

```
   task b_transport_2(int id=-1, my_trans trans,
                      ref int delay);
   endtask
endclass
```

# vmm_tlm_transport_interconnect#(DATA)

Interconnect transport class.

Class extended from `vmm_tlm_transport_interconnect_base`. This class is specific to `vmm_tlm::phase_e` type.

The parameter DATA is the type of the transaction object of the port/export services. The default type is `vmm_data`.

## SystemVerilog

```
class vmm_tlm_transport_interconnect #(type DATA = vmm_data)
extends vmm_tlm_transport_interconnect_base#(DATA);
```

### Description

Used to connect `vmm_tlm` port to a non-matching export.

## Summary


- vmm_tlm_transport_interconnect::new() page B-364

## vmm_tlm_transport_interconnect::new()

Constructor of an interconnect class.

### SystemVerilog

```
function new(vmm_object parent,string name);
```

### Description

Sets the parent, if it is an extension of `vmm_object`. Sets the name of the instance.

# vmm_tlm_transport_interconnect_base#(DATA,PHASE)

Interconnect transport base class.

Base class for `vmm_tlm_transport_interconnect` class. This class contains `tlm_bind` method which is used to connect below ports and exports.

- `vmm_tlm_b_transport_port to vmm_tlm_nb_transport_export`
- `vmm_tlm_b_transport_port to vmm_tlm_nb_transport_fw_export`
- `vmm_tlm_nb_transport_port to vmm_tlm_b_transport_export`
- `vmm_tlm_nb_transport_fw_port to vmm_tlm_b_transport_export`

Any user-defined interconnect class should be extended from this base class. The parameter DATA is the type of the transaction object of the port/export services. The default type is `vmm_data`. The parameter PHASE is the type of the phasing class. The default value is `vmm_tlm::phase_e`.

## SystemVerilog

```
class vmm_tlm_transport_interconnect_base #(type DATA =
vmm_data , type PHASE = vmm_tlm::phase_e) extends vmm_object;
```

## Description

Used to connect `vmm_tlm` port to a non-matching export.

## Summary


- vmm_tlm_transport_interconnect_base::new() ..... page B-367
- vmm_tlm_transport_interconnect_base::tlm_bind() .. page B-368

## vmm_tlm_transport_interconnect_base::new()

Constructor of an interconnect base class.

### SystemVerilog

```
function new(vmm_object parent, string name);
```

### Description

Sets the parent, if it is an extension of `vmm_object`. Sets the name of the instance.

## vmm_tlm_transport_interconnect_base::tlm_bind()

Binds the TLM port to TLM export.

### SystemVerilog

```
function int tlm_bind(vmm_tlm_base tlm_intf_port,
vmm_tlm_base tlm_intf_export, vmm_tlm::intf_e intf, string
fname = "", int lineno = 0);
```

### Description

Binds the `tlm_intf_port to tlm_intf_export`, which are passed as arguments to the function.

First argument to the function is tlm port and the second argument is tlm export. If wrong types are passed to first or second argument then an error is issued.

Third argument takes type of the non-blocking port or export.

- `vmm_tlm::TLM_NONBLOCKING_EXPORT`

  This is used when producer is `vmm_tlm_b_transport_port` and consumer is `vmm_tlm_nb_transport_export`.

- `vmm_tlm::TLM_NONBLOCKING_FW_EXPORT`

  This is used when producer is `vmm_tlm_b_transport_port` and consumer is `vmm_tlm_nb_transport_fw_export.`

- `vmm_tlm::TLM_NONBLOCKING_PORT`

  This is used when producer is `vmm_tlm_nb_transport_port` and consumer is `vmm_tlm_b_transport_export.`

- `vmm_tlm::TLM_NONBLOCKING_FW_PORT`

  This is used when producer is `vmm_tlm_nb_transport_fw_port` and consumer is `vmm_tlm_b_transport_export`.

Any other values for third argument will issue an error.

# vmm_tlm_reactive_if #(DATA, q_size)

TLM Reactive class providing an API similar to the `vmm_channel's` active slot.

## SystemVerilog

```
class vmm_tlm_reactive_if#(type DATA = vmm_data, int q_size
= 1) extends vmm_object;
```

## Description

It facilitates writing reactive transactors using a polling approach rather than an interrupt approach. It provides blocking, non-blocking_fw and non-blocking (bi-directional) exports and can be bound to more than one port.

## Summary


- vmm_tlm_reactive_if::completed() ..... page B-371
- vmm_tlm_reactive_if::get() ..... page B-372
- vmm_tlm_reactive_if::new() ..... page B-373
- vmm_tlm_reactive_if::tlm_bind() ..... page B-374
- vmm_tlm_reactive_if::try_get() ..... page B-376

## vmm_tlm_reactive_if::completed()

Indicate that the previously activated transaction has been completed.

### SystemVerilog

```
function void completed();
```

### Description

The completed method must be called by the transactor to indicate the completion of active transaction. The blocking port which initiated the transaction will be unblocked and `nb_transport_bw` method is called for non-blocking bi-directional with `TLM_COMPLETED` phase.

For `vmm_data` derivatives `vmm_data::ENDED` is also indicated. The transaction is removed from the pending queue only when completed is called.

### Example

```
class consumer extends vmm_xactor;
  vmm_tlm_reactive_if#(my_trans, 4) reac_export1 = new(this,
"export1");
   virtual task run_ph();
      my_trans trans;
      fork
        while (1)
        begin
            reac_export1.get(trans);
            reac_export1.completed();
        end
      join_none
   endtask : run_ph
endclass : consumer
```

## vmm_tlm_reactive_if::get()

Blocking method  to get the next transaction object.

### SystemVerilog

```
task get(output DATA tr);
```

### Description

Blocks until a transaction object is available. If there is more than one object then gets the first transaction object. Subsequent get calls must be preceded by calling the `completed()` method. Else, an error is issued.

### Example

```
class consumer extends vmm_xactor;
  vmm_tlm_reactive_if#(my_trans, 4) reac_export1 = new(this,
"export1");
   virtual task run_ph();
      my_trans trans;
      fork
        while (1)
        begin
            reac_export1.get(trans);
            reac_export1.completed();
        end
      join_none
   endtask : run_ph
endclass : consumer
Indicate that the previously activated transaction has been
completed.
```

## vmm_tlm_reactive_if::new()

Constructor of reactive interface class.

### SystemVerilog

```
function new(vmm_object parent, string name);
```

### Description

Sets the parent, if it is an extension `vmm_object`. Sets the name of the instance.

## vmm_tlm_reactive_if::tlm_bind()

Binds the TLM port passed as an argument to the corresponding TLM export depending on the enum passed in second argument.

### SystemVerilog

```
function int tlm_bind(vmm_tlm_base tlm_intf,
vmm_tlm::intf_e intf);
```

### Description

Binds the TLM port passed as an argument to one of the export in the class depending on the enum value passed as second arguement.

The second argument can be,

- `vmm_tlm::TLM_NONBLOCKING_EXPORT`

  Port passed as an argument is connected to `vmm_tlm_nb_transport_export` (bi-directional)

- `vmm_tlm::TLM_BLOCKING_EXPORT`

  Port passed as an argument is connected to `vmm_tlm_b_transport_export`

- `vmm_tlm::TLM_NONBLOCKING_FW_EXPORT`

  Port passed as an argument is connected to `vmm_tlm_nb_transport_fw_export` (forward only)

## Example

```
class consumer extends vmm_xactor;
  vmm_tlm_reactive_if#(my_trans, 4) reac_export1 = new(this,
"export1");
   virtual task run_ph();
      my_trans trans;
      fork
        while (1)
        begin
            reac_export1.get(trans);
            reac_export1.completed();
        end
      join_none
   endtask : run_ph
endclass : consumer
class producer extends vmm_xactor;
   vmm_tlm_b_transport_port#(producer) b_port = new(this,
"producer port");
endclass
class my_env extends vmm_group;
  producer p1;
  producer p2;
  consumer c;
  function void connect_ph();
      c.reac_export1.tlm_bind(p1.b_port,
vmm_tlm::TLM_BLOCKING_EXPORT);
       c.reac_export1.tlm_bind(p2.b_port,
vmm_tlm::TLM_BLOCKING_EXPORT);
  endfunction
endclass
```

## vmm_tlm_reactive_if::try_get()

Non-blocking function  to get the next transaction object.

### SystemVerilog

```
Function DATA try_get();
```

### Description

Returns null if no transaction object is received. If there are more than one object then returns the first transaction object. Subsequent `try_get` calls must be preceded by calling the `completed()` method. Else, an error is issued.

### Example

```
class consumer extends vmm_xactor;
  vmm_tlm_reactive_if#(my_trans, 4) reac_export1 = new(this,
"export1");
   virtual task run_ph();
      my_trans trans;
      fork
        while (1)
        begin
            trans = reac_export1.try_get();
            reac_export1.completed();
        end
      join_none
   endtask : run_ph
endclass : consumer
```

# vmm_unit

Base class for providing pre-defined simulation phases.

## SystemVerilog

```
virtual class vmm_unit extends vmm_object;
```

## Description

This class is used as the base class that provides pre-defined simulation phases to structural elements, such as transactors, transaction-level models and generators. The purpose of this class is to:

- Support structural composition and connectivity.
- Integrate into a simulation timeline.

The `vmm_unit` class should not be directly extended. Instead, `vmm_xactor` and `vmm_group` are extended from `vmm_unit`.

Since the `vmm_xactor` and `vmm_group` base classes are extended from `vmm_unit`, all classes extended from these base classes can invoke the phase methods. You should continue to extend `vmm_xactor` for implementing transactors, and for implementing compositions (combination of components), extend `vmm_group`.

The following are the phases, listed in the order in which they are called:

- `build_ph()`
- `configure_ph()`

- connect_ph()
- configure_test_ph()
- start_of_sim_ph()
- reset_ph()
- training_ph()
- config_dut_ph()
- start_ph()
- start_of_test_ph()
- run_ph()
- shutdown_ph()
- cleanup_ph()
- report_ph()
- final

## Summary


- vmm_unit::build_ph() ..... page B-380
- vmm_unit::cleanup_ph() ..... page B-381
- vmm_unit::config_dut_ph() ..... page B-382
- vmm_unit::configure_ph() ..... page B-383
- vmm_unit::connect_ph() ..... page B-384
- vmm_unit::consensus_requested() ..... page B-385
- vmm_unit::consent() ..... page B-386
- vmm_unit::disabled_ph() ..... page B-387
- vmm_unit::disable_unit() ..... page B-388
- vmm_unit::forced() ..... page B-390
- vmm_unit::force_thru() ..... page B-391
- vmm_unit::get_timeline() ..... page B-392
- vmm_unit::is_unit_enabled() ..... page B-393
- vmm_unit::new() ..... page B-394
- vmm_unit::oppose() ..... page B-395
- vmm_unit::override_phase() ..... page B-396
- vmm_unit::report_ph() ..... page B-397
- vmm_unit::request_consensus() ..... page B-398

- vmm_unit::reset_ph() ..... page B-399
- vmm_unit::run_ph() ..... page B-400
- vmm_unit::shutdown_ph() ..... page B-401
- vmm_unit::start_of_sim_ph() ..... page B-402
- vmm_unit::start_of_test_ph() ..... page B-403
- vmm_unit::start_ph() ..... page B-404
- vmm_unit::training_ph() ..... page B-405

## vmm_unit::build_ph()

Method to build this component.

### SystemVerilog

```
virtual function void vmm_unit::build_ph();
```

### Description

Builds this component. Leaf level or independent root components associated can be created here.

### Example

```
class memsys_env extends vmm_group;
   cpu_subenv extends cpu0;
   vmm_ms_scenario_gen gen;
   memsys_scenario memsys_scn;
   ...
   function void build_ph();
      cpu0 = new("subenv", "CPU0", this);
      cpu1 = new("subenv", "CPU1", this);
      memsys_scn = new();
      gen = new("MS-Generator");
      ...
   endfunction
endclass
```

## vmm_unit::cleanup_ph()

Method for post-execution.

### SystemVerilog

```
virtual task vmm_unit::cleanup_ph();
```

### Description

Method to perform post-execution verification, if it is enabled.

### Example

```
class groupExtension extends vmm_group;
   task cleanup_ph();
      `vmm_note(log,`vmm_sformatf(
         "groupExtension::cleanup_ph"));
      ...
   endtask:cleanup_ph
endclass
```

## vmm_unit::config_dut_ph()

Method for DUT configuration.

### SystemVerilog

```
virtual task vmm_unit::config_dut_ph();
```

### Description

Initialization of  the DUT attached to this component, if it is enabled.

### Example

```
class vdmsys_env extends vmm_group;
   task config_dut_ph;
      top.write_reg(N_RD_PORT, 20);
      top.write_reg(N_WR_PORT, 30);
      ...
   endtask
endclass
```

## vmm_unit::configure_ph()

Method for functional configuration.

### SystemVerilog

```
virtual function void vmm_unit::configure_ph();
```

### Description

Functional configuration of this component.

### Example

```
class groupExtension extends vmm_group;
   ...
   function void configure_ph();
      `vmm_note
      (log,`vmm_sformatf("groupExtension::configure_ph"));
      ...
   endfunction:configure_ph
endclass
```

## vmm_unit::connect_ph()

Method for connecting components.

### SystemVerilog

```
virtual function void vmm_unit::connect_ph();
```

### Description

Connects the interfaces that are wholly contained within this component.

### Example

```
class memsys_env extends vmm_group;
   cpu_subenv extends cpu0;
   vmm_ms_scenario_gen gen;
   memsys_scenario memsys_scn;
   ...
   function void build_ph();
      cpu0 = new("subenv", "CPU0", this);
      cpu1 = new("subenv", "CPU1", this);
      memsys_scn = new();
      gen = new("MS-Generator");
      ...
   endfunction

   function void memsys_env::connect_ph();
      gen.register_channel("cpu0_chan",
         cpu0.gen_to_drv_chan);
      gen.register_channel("cpu1_chan",
         cpu1.gen_to_drv_chan);
      gen.register_ms_scenario( "memsys_scn", memsys_scn);
      ...
   endfunction
endclass
```

## vmm_unit::consensus_requested()

A consensus request is made.

### SystemVerilog

```
virtual function void consensus_requested(vmm_unit who);
```

### OpenVera

```
Not supported
```

### Description

When this method is called, it indicates that a consensus request is made to this currently-opposing unit by the specified unit, by calling the `vmm_unit::request_consensus()` method.

This method should be extended, if this unit is to honor consensus requests.

## vmm_unit::consent()

Expresses the consent of this `vmm_unit` to the consensus for the specified reason.

### SystemVerilog

```
function void vmm_unit::consent(string why =
   "No reason specified");
```

### Description

Expresses the consents of this `vmm_unit` to the consensus for the specified reason.

### Example

```
class groupExtension extends vmm_group;
   ...
   task reset_ph();
      this.oppose("reset phase running");
      fork
         begin
            #50;
            this.consent("reset phase finished");
         end
      join_none
   endtask:reset_ph
   ...
endclass
```

## vmm_unit::disabled_ph()

Method executes instead of the `reset_ph()` method, when unit disabled.

### SystemVerilog

```
virtual task vmm_unit::disabled_ph();
```

### Description

This Method gets executed instead of the r`eset_ph()` method, if this `vmm_unit` instance is disabled.

### Example

```
class groupExtension extends vmm_group;
   function void disabled_ph();
      `vmm_note(log,`vmm_sformatf(
         "groupExtension::disabled_ph"));
      ...
   endfunction:disabled_ph
endclass
```

## vmm_unit::disable_unit()

Disables a unit instance.

### SystemVerilog

```
function void vmm_unit::disable_unit();
```

### Description

Disables this instance of the `vmm_unit` class. This method must be called, before the `start_of_sim` phase. A `vmm_unit` instance can only be re-enabled by resetting its timeline to the `configure` phase or earlier.

### Example

```
class groupExtension extends vmm_group;
   ...
endclass

groupExtension m1 = new ("groupExtension","m1");
m1.disable_unit();
```

## vmm_unit::final_ph()

Method to publish final report.

### SystemVerilog

```
function void vmm_unit::final_ph();
```

### Description

In case of multiple concatenated tests, final phase can be used to summarize the final report.

### Example

```
class testExtension extends vmm_test;
…….
  function void final_ph();
    env.summary();
  endfunction

endclass
```

## vmm_unit::forced()

Forces consensus on this unit.

### SystemVerilog

```
function void forced(string why = "No reason specified");
```

### OpenVera

```
Not supported
```

### Description

Forces consensus for this unit to be reached. The consensus may be subsequently consented to by calling the `vmm_unit::consent()` method, or it may be opposed by calling the `vmm_unit::oppose()` method.

The forcing of consensus through the parent unit occurs, only if this unit is configured to force through to its parent by the `vmm_unit::force_thru()` method. The `why` argument is a string that specifies the reason why the consensus is forced on this unit.

## vmm_unit::force_thru()

Forces sub-consensus from a sub-unit through or not.

### SystemVerilog

```
function void force_thru(vmm_unit child, bit thru = 1);
```

### OpenVera

```
Not supported
```

### Description

If the “thru” argument is TRUE, any consensus forced on the specified child unit instance will force the consensus on this unit instance.

If the “thru” argument is FALSE, any consensus forced on the specified child unit instance will simply consent to the consensus on this unit instance.

## vmm_unit::get_timeline()

Returns the enclosing timeline.

### SystemVerilog

```
function vmm_timeline vmm_unit::get_timeline();
```

### Description

Returns the runtime timeline, this unit is executing under.

### Example

```
class groupExtension extends vmm_group;
   ...
   function void build_ph();
      vmm_timeline t = this.get_timeline();
      ...
   endfunction
endclass
```

## vmm_unit::is_unit_enabled()

Returns 1, if unit is enabled.

### SystemVerilog

```
function bit vmm_unit::is_unit_enabled();
```

### Description

Checks if this `vmm_unit` instance is disabled or not. By default, all units are enabled. A unit may be disabled by calling its `disable_unit()` method, before the `start_of_sim` phase.

### Example

```
class groupExtension extends vmm_group;
   ...
endclass

class udf_start_def extends vmm_fork_task_phase_def
      #(groupExtension);
   ...
   task do_task_phase(groupExtension obj);
      if(obj.is_unit_enabled())
         obj.udf_start_ph();
   endtask:do_task_phase
   ...
endclass
```

## vmm_unit::new()

Constructor for the `vmm_unit`.

### SystemVerilog

```
function vmm_unit::new(string name, string inst,
   vmm_object parent = null);
```

### Description

Constructs an instance of this class with the specified name, instance name, and optional parent.

The specified name is used as the name of the embedded `vmm_log`. The specified instance name is used as the name of the underlying `vmm_object`.

### Example

```
class vip1 extends vmm_group;
   function new (string name, string inst);
      super.new (name, inst, this);
   endfunction
endclass
```

## vmm_unit::oppose()

Expresses the opposition of this `vmm_unit` to the consensus for the specified reason.

### SystemVerilog

```
function void vmm_unit::oppose(string why =
   "No reason specified");
```

### Description

Expresses the opposition of this `vmm_unit` to the consensus for the specified reason.

### Example

```
class groupExtension extends vmm_group;
   ...
   task reset_ph();
      this.oppose("reset phase running");
      fork
         begin
            #50;
            this.consent("reset phase finished");
         end
      join_none
   endtask:reset_ph
   ...
endclass
```

## vmm_unit::override_phase()

Method to execute new phase definition instead of the existing one.

### SystemVerilog

```
virtual function vmm_phase_def
vmm_unit::override_phase(string name, vmm_phase_def def);
```

### Description

Overrides the specified phase with the specified phase definition for this instance. If *def* is null, the override (if any) is removed. Returns the previous override phase definition (if any).

### Example

```
class cust_configure_phase_def #(type T = groupExtension)
      extends vmm_topdown_function_phase_def #(T);
   function void do_function_phase( T obj);
      obj.cust_config_ph();
   endfunction
endclass

class groupExtension extends vmm_group;
   function void config_ph();
      `vmm_note(log,`vmm_sformatf(
         "groupExtension::configure_ph"));
    endfunction:config_ph
   function void cust_config_ph();
      `vmm_note(log,`vmm_sformatf(
         "groupExtension::cust_config_ph"));
   endfunction:cust_config_ph
endclass

cust_configure_phase_def  cust_cfg = new();
groupExtension   m1 = new("groupExtension","m1");
`void(m1.override_phase("configure",cust_cfg ));
```

## vmm_unit::report_ph()

Method for test reporting.

### SystemVerilog

```
virtual function void vmm_unit::report_ph();
```

### Description

Method to perform post-test pass or fail reporting, if it is enabled.

### Example

```
class memsys_env extends vmm_group;
   function void report_ph();
      sb.report;
      ...
   endfunction
endclass
```

## vmm_unit::request_consensus()

Requests that a consensus be reached.

### SystemVerilog

```
task request_consensus(string why = "No reason specified");
```

### OpenVera

```
Not supported
```

### Description

Makes a request of all currently-opposing participants in this unit instance that they consent to the consensus.

A request is made by calling the `vmm_unit::consensus_requested()` method in this unit, and all currently-opposing child units. If a forced consensus on this unit forces through to a higher-level unit, then the consensus request is propagated upward as well. This task returns when the local unit-level consensus is reached.

The `why` argument is a string that specifies the reason why the consensus is forced on this unit.

## vmm_unit::reset_ph()

Method for reset.

### SystemVerilog

```
virtual task vmm_unit::reset_ph()
```

### Description

Resets this unit, if it is enabled. This method is executed at the reset phase.

### Example

```
class memsys_env extends vmm_group;
   task reset_ph();
      // Resetting the DUT
      test_top.reset <= 1'b0;
      repeat(1) @(test_top.port0.cb)
      test_top.reset <= 1'b1;
      repeat(10) @(test_top.port0.cb)
      test_top.reset <= 1'b0;
      `vmm_verbose(this.log,"RESET DONE...");
   endtask
endclass
```

## vmm_unit::run_ph()

Body of test, if it is enabled.

### SystemVerilog

```
virtual task vmm_unit::run_ph();
```

### Description

Body of test, if it is enabled. Can be interrupted by resetting this component. May be stopped.

### Example

```
class groupExtension extends vmm_group;
   task run_ph();
      `vmm_note(log,`vmm_sformatf(
         "groupExtension::run_ph"));
      ...
   endtask : run_ph
endclass
```

## vmm_unit::shutdown_ph()

Method to stop all unit components.

### SystemVerilog

```
virtual task vmm_unit::shutdown_ph();
```

### Description

Method to stop processes within this component, if it is enabled.

### Example

```
class cpu_subenv extends vmm_group;
   ...
   task shutdown_ph();
      if (enable_gen) this.gen.stop_xactor();
   endtask
   ...
endclass
```

## vmm_unit::start_of_sim_ph()

Method executes at start of simulation.

### SystemVerilog

```
virtual function void vmm_unit::start_of_sim_ph();
```

### Description

Method called at start of the simulation.

### Example

```
class cpu_driver extends vmm_group;
   ...
   function void start_of_sim_ph();
      if (iport == null)
         `vmm_fatal(log, "Virtual port not connected to the
           actual interface instance");
   endfunction
   ...
endclass
```

## vmm_unit::start_of_test_ph()

Method called at start of the test body.

### SystemVerilog

```
virtual function void vmm_unit::start_of_test_ph();
```

### Description

Method called at start of the test body, if it is enabled.

### Example

```
class groupExtension extends vmm_group;
   function void start_of_test_ph();
      `vmm_note(log,`vmm_sformatf(
         "groupExtension::start_of_test_ph"));
      ...
   endfunction:start_of_test_ph
endclass
```

## vmm_unit::start_ph()

Method to start unit components.

### SystemVerilog

```
virtual task vmm_unit::start_ph();
```

### Description

Method to start processes within this component, if it is enabled.

### Example

```
class memsys_env extends vmm_group;
   ...
   task start_ph();
      this.gen.start_xactor();
   endtask
   ...
endclass
```

## vmm_unit::training_ph()

Method for training.

### SystemVerilog

```
virtual task vmm_unit::training_ph();
```

### Description

Initialization of this component, such as interface training.

### Example

```
class groupExtension extends vmm_group;
   task training_ph();
      `vmm_note(log,`vmm_sformatf(
         "groupExtension::training_ph"));
      ...
   endtask:training_ph
endclass
```

# vmm_version

This class is used to report the version and vendor of the VMM Standard Library implementation.

## Summary


- vmm_version::display() ..... page B-407
- vmm_version::major() ..... page B-408
- vmm_version::minor() ..... page B-409
- vmm_version::patch() ..... page B-410
- vmm_version::psdisplay() ..... page B-411
- vmm_version::vendor() ..... page B-412

## vmm_version::display()

Displays the version.

### SystemVerilog

```
function void display(string prefix = "");
```

### OpenVera

Not supported.

### Description

Displays the version image returned by the **`psdisplay()`** method, to the standard output.

The argument `prefix` is used to append a string to the content displayed by this method.

## vmm_version::major()

Returns the major revision number.

### SystemVerilog

```
function int major();
```

### OpenVera

Not supported.

### Description

Returns the major version number of the implemented VMM Standard Library. Should always return 1.

## vmm_version::minor()

Returns the minor revision number.

### SystemVerilog

```
function int minor();
```

### OpenVera

```
function integer minor();
```

### Description

Returns the minor version number of the implemented VMM Standard Library.  Should always return 5, if the additions and updates specified in this appendix are fully implemented.

### Example

*Example B-132*

```
initial begin
   string minor_ver;
   vmm_version v = new;
   $sformat(minor_ver,"VMM Minor Version %d", v.minor());
   `vmm_note(log,minor_ver);
end
```

## vmm_version::patch()

Returns the patch number.

### SystemVerilog

```
function int patch();
```

### OpenVera

Not supported.

### Description

Returns the patch number of the implemented VMM Standard Library. The returned value is vendor-dependent.

## vmm_version::psdisplay()

Formats the major and minor version, patch, and vendor information.

### SystemVerilog

```
function string psdisplay(string prefix = "");
```

### OpenVera

Not supported.

### Description

Creates a well formatted image of the VMM Standard Library implementation version information. The format is:

*prefix* `VMM Version` *major*`.`*minor*`.`*patch* `(`*vendor*`)`

## vmm_version::vendor()

Returns the name of the library vendor.

### SystemVerilog

```
function string vendor();
```

### OpenVera

Not supported.

### Description

Returns the name of the vendor supplying the VMM Standard Library implementation. The returned value is vendor-dependent.

# vmm_voter

This class is an interface to participate in a consensus, and indicates consent or opposition to the end of test. It is created through the "vmm_consensus::register_voter()" method. Its constructor is not documented, therefore, it must not be created directly.

## Summary


- vmm_voter::consent() ..... page B-414
- vmm_voter::forced() ..... page B-415
- vmm_voter::oppose() ..... page B-416

## vmm_voter::consent()

Agrees to a consensus.

### SystemVerilog

```
function void consent(string why = "No specified reason");
```

### OpenVera

```
task consent(string why = "No specified reason");
```

### Description

Allows consensus to be reached for the optionally specified reason. This method may be called repeatedly to modify the reason for the consent. A consent may be withdrawn by calling the "vmm_voter::oppose()" method.

### Example

*Example B-133*

```
program test_consensus;

   string who[];
   string why[];
   vmm_consensus vote = new("Vote", "Main");
   vmm_voter v1;

   initial begin
      v1 = vote.register_voter("Voter #1");
      v1.consent("Consent by default");
      ...
   end

endprogram
```

## vmm_voter::forced()

Forces a consensus.

### SystemVerilog

```
function void forced(string why = "No specified reason");
```

### OpenVera

```
task forced(string why = "No specified reason");
```

### Description

Forces an end of test consensus for the optionally specified reason. The end of test is usually forced by a directed testcase, but can be forced by any participant, as necessary. A forced consensus may be cancelled (if the simulation is still running) by calling the "vmm_voter::oppose()" or "vmm_voter::consent()" method.

### Example

*Example B-134*

```
initial begin
   ...
   vmm_voter test_voter = env.end_vote.register_voter(
      "Test case Stimulus");
   test_voter.oppose("Test not done");
   ...
   test_voter.forced("Test is done");
end
```

## vmm_voter::oppose()

Opposes to a consensus.

### SystemVerilog

```
function void oppose(string why = "No specified reason");
```

### OpenVera

```
task oppose(string why = "No specified reason");
```

### Description

Prevents consensus from being reached for the optionally specified reason, by default. This method may be called repeatedly to modify the reason for the opposition.

### Example

*Example B-135*

```
initial begin
   my_env env = new();
   vmm_voter test_voter = env.end_vote.register_voter(
      "Test case Stimulus");
   test_voter.oppose("test not done");
end
```

# vmm_xactor

This base class is to be used as the basis for all transactors, including bus-functional models, monitors, and generators. It provides a standard control mechanism expected in all transactors.

## Summary


- vmm_xactor::append_callback() ..... page B-418
- vmm_xactor::do_psdisplay() ..... page B-419
- vmm_xactor::do_reset_xactor() ..... page B-420
- vmm_xactor::do_start_xactor() ..... page B-422
- vmm_xactor::do_stop_xactor() ..... page B-424
- vmm_xactor::do_what_e ..... page B-426
- vmm_xactor::exp_vmm_sb_ds() ..... page B-428
- vmm_xactor::get_input_channels() ..... page B-429
- vmm_xactor::get_instance() ..... page B-430
- vmm_xactor::get_name() ..... page B-431
- vmm_xactor::get_output_channels() ..... page B-432
- vmm_xactor::inp_vmm_sb_ds() ..... page B-433
- vmm_xactor::kill() ..... page B-434
- vmm_xactor::log ..... page B-435
- vmm_xactor::main() ..... page B-436
- vmm_xactor::new() ..... page B-437
- vmm_xactor::notifications_e ..... page B-438
- vmm_xactor::notify ..... page B-440
- vmm_xactor::prepend_callback() ..... page B-442
- vmm_xactor::psdisplay() ..... page B-444
- vmm_xactor::register_vmm_sb_ds() ..... page B-445
- vmm_xactor::reset_xactor() ..... page B-446
- vmm_xactor::restore_rng_state() ..... page B-449
- vmm_xactor::stream_id ..... page B-450
- vmm_xactor::save_rng_state() ..... page B-451
- vmm_xactor::start_xactor() ..... page B-452
- vmm_xactor::stop_xactor() ..... page B-453
- vmm_xactor::unregister_callback() ..... page B-454
- vmm_xactor::unregister_vmm_sb_ds() ..... page B-455
- vmm_xactor::`vmm_callback() ..... page B-456
- vmm_xactor::wait_if_stopped() ..... page B-457
- vmm_xactor::wait_if_stopped_or_empty() ..... page B-459
- vmm_xactor::xactor_status() ..... page B-461
- `vmm_xactor_member_begin() ..... page B-462
- `vmm_xactor_member_end() ..... page B-463
- `vmm_xactor_member_scalar*() ..... page B-464
- `vmm_xactor_member_string*() ..... page B-466
- `vmm_xactor_member_enum*() ..... page B-468
- `vmm_xactor_member_vmm_data*() ..... page B-470
- ``vmm_xactor_member_channel*() ..... page B-472
- `vmm_xactor_member_xactor*() ..... page B-474
- `vmm_xactor_member_user_defined() ..... page B-476

## vmm_xactor::append_callback()

Appends the specified callback façade instance with this instance of the transactor.

### SystemVerilog

```
virtual function void
           append_callback(vmm_xactor_callbacks cb);
```

### OpenVera

Not supported.

### Description

Callback methods are invoked in the order in which they were registered.

A warning is generated, if the same callback façade instance is registered more than once with the same transactor. A façade instance can be registered with more than one transactor. Callback façade instances can be unregistered and re-registered dynamically.

## vmm_xactor::do_psdisplay()

Overrides the shorthand `psdisplay()` method.

### SystemVerilog

```
virtual function string do_psdisplay(string prefix = "")
```

### OpenVera

Not supported.

### Description

This method overrides the default implementation of the `vmm_xactor::psdisplay()` method, created by the `vmm_xactor` shorthand macros. If defined, it will be used instead of the default implementation.

### Example

*Example B-136*

```
class eth_frame_gen extends vmm_xactor;
   ...
   `vmm_xactor_member_begin(eth_frame_gen)
   ...
   `vmm_xactor_member_end(eth_frame_gen)
    virtual function string do_psdisplay(string prefix = "")
      $sformat(do_psdisplay,"%s Printing Ethernet frame \n
          generator members \n",prefix);
      ...
   endfunction
   ...
endclass
```

## vmm_xactor::do_reset_xactor()

Overrides the shorthand `reset_xactor()` method.

### SystemVerilog

```
protected virtual function void do_reset_xactor(
    vmm_xactor::reset_e rst_typ)
```

### OpenVera

Not supported.

### Description

Overrides the default implementation of the **`vmm_xactor::reset_xactor()`** method created by the **`vmm_xactor`** shorthand macros. If defined, it is used instead of the default implementation.

### Example

*Example B-137*

```
class xact1 extends vmm_xactor;
   ...
endclass

class xact2 extends vmm_xactor;
   ...
endclass

class xact extends vmm_xactor;
   xact1 xact1_inst;
   xact2 xact2_inst;
   ...
`vmm_xactor_member_begin(xact)
```

```
   vmm_xactor_member_xactor(xact1_inst,DO_ALL)
   vmm_xactor_member_xactor(xact2_inst,DO_ALL)
`vmm_xactor_member_end(xact)
   protected virtual function void do_reset_xactor ();
     `ifdef XACT_2
        xact2_inst.reset_xactor();
     `else
      xact1_inst.reset_xactor();
     `endif
     ...
   endfunction
   ...
endclass
```

## vmm_xactor::do_start_xactor()

Overrides the shorthand `start_xactor()` method.

### SystemVerilog

```
protected virtual function void do_start_xactor()
```

### OpenVera

Not supported.

### Description

Overrides the default implementation of the `vmm_xactor::start_xactor()` method, created by the `vmm_xactor` shorthand macros. If defined, it is used instead of the default implementation.

### Example

*Example B-138*

```
class xact1 extends vmm_xactor;
   ...
endclass

class xact2 extends vmm_xactor;
   ...
endclass

class xact extends vmm_xactor;
   xact1 xact1_inst;
   xact2 xact2_inst;
   ...
`vmm_xactor_member_begin(xact)
   vmm_xactor_member_xactor(xact1_inst,DO_ALL)
```

```
   vmm_xactor_member_xactor(xact2_inst,DO_ALL)
`vmm_xactor_member_end(xact)
   protected virtual function void do_start_xactor ();
     `ifdef XACT_2
       xact2_inst.start_xactor();
      `else
      xact1_inst.start_xactor();
      `endif
   ...
   endfunction
   ...
endclass
```

## vmm_xactor::do_stop_xactor()

Overrides the shorthand `stop_xactor()` method.

### SystemVerilog

```
protected virtual function void do_stop_xactor()
```

### OpenVera

Not supported.

### Description

This method overrides the default implementation of the `vmm_xactor::stop_xactor()` method, created by the `vmm_xactor` shorthand macros. If defined, it will be used instead of the default implementation.

### Example

*Example B-139*

```
class xact1 extends vmm_xactor;
   ...
endclass

class xact2 extends vmm_xactor;
   ...
endclass

class xact extends vmm_xactor;
   xact1 xact1_inst;
   xact2 xact2_inst;
   ...
`vmm_xactor_member_begin(xact)
   vmm_xactor_member_xactor(xact1_inst,DO_ALL)
```

```
   vmm_xactor_member_xactor(xact2_inst,DO_ALL)
`vmm_xactor_member_end(xact)
   protected virtual function void do_stop_xactor ();
     `ifdef XACT_2
       xact2_inst.stop_xactor();
      `else
      xact1_inst.stop_xactor();
      `endif
      ...
   endfunction
   ...
endclass
```

## vmm_xactor::do_what_e

Specifies which methods are to be provided by a shorthand implementation.

### SystemVerilog

```
typedef enum {DO_PRINT   ='h001,
              DO_START   ='h002,
              DO_STOP    ='h004,
              DO_RESET   ='h010,
              DO_KILL    ='h020,
              DO_ALL     ='hFFF} do_what_e;
```

### OpenVera

Not supported.

### Description

Used to specify which methods are to include the specified data members in their default implementation. The `"DO_PRINT"` includes the member in the default implementation of the `psdisplay()` method. The `"DO_START"` includes the member in the default implementation of the `start_xactor()` method, if applicable. The `"DO_STOP"` includes the member in the default implementation of the `stop_xactor()` method, if applicable. The `"DO_RESET"` includes the member in the default implementation of the `reset_xactor()` method, if applicable.

Multiple methods can be specified by adding or using `or` in the individual symbolic values. All methods are specified by providing the `"DO_ALL"` symbol.

## Example

*Example B-140*

```
`vmm_xactor_member_xactor(idler, DO_ALL - DO_STOP);
```

## vmm_xactor::exp_vmm_sb_ds()

For more information on this method, refer to the *VMM Scoreboard User Guide*.

## vmm_xactor::get_input_channels()

Returns the input channels of this transactor.

### SystemVerilog

```
function void get_input_channels(ref vmm_channel chans[$]);
```

### OpenVera

Not supported.

### Description

Returns the channels where this transactor is identified as the consumer using the `vmm_channel::set_consumer()` method.

### Example

*Example B-141*

```
class xactor extends vmm_xactor;
   ...
endclass

program prog;
   xactor xact = new;
   vmm_channel in_chans[$];
   ...
   initial begin
      ...
      xact.get_input_channels(in_chans);
      ...
   end

endprogram
```

## vmm_xactor::get_instance()

Returns the instance name of this transactor.

### SystemVerilog

```
virtual function string get_instance();
```

### OpenVera

Not supported.

## vmm_xactor::get_name()

Returns the name of this transactor.

### SystemVerilog

```
virtual function string get_name();
```

### OpenVera

Not supported.

## vmm_xactor::get_output_channels()

Returns the output channels of this transactor.

### SystemVerilog

```
function void get_output_channels(
    ref vmm_channel chans[$]);
```

### OpenVera

Not supported.

### Description

Returns the channels where this transactor is identified as the producer, using the `vmm_channel::set_producer()` method.

### Example

*Example B-142*

```
class xactor extends vmm_xactor;
   ...
endclass

program prog;
   xactor xact = new;
   vmm_channel out_chans[$];
   ...
   initial begin
      ...
      xact.get_output_channels (out_chans);
      ...
   end

endprogram
```

## vmm_xactor::inp_vmm_sb_ds()

For more information on this method, refer to the *VMM Scoreboard User Guide*.

## vmm_xactor::kill()

Prepares a transactor for deletion.

### SystemVerilog

```
function void kill();
```

### OpenVera

Not supported.

### Description

Prepares a transactor for deletion and reclamation by the garbage collector.

Removes this transactor as the producer of its output channels, and as the consumer of its input channels. De-registers all data stream scoreboards and callback extensions.

### Example

*Example B-143*

```
class xactor extends vmm_xactor;
   ...
endclass
program prog;
   xactor xact = new;
   ...
   initial begin
      xact.kill();
      ...
   end
endprogram
```

## vmm_xactor::log

Message service interface for messages, which are generated from within this transactor instance.

### SystemVerilog

```
vmm_log log;
```

### OpenVera

Not supported.

## vmm_xactor::main()

Forks-off this task whenever the **`start_xactor()`** method is called.

### SystemVerilog

```
protected virtual task main();
```

### OpenVera

Not supported.

### Description

This task is forked off, whenever the **`start_xactor()`** method is called. It is terminated, whenever the **`reset_xactor()`** method is called. The functionality of a user-defined transactor must be implemented in this method. Any additional subthreads must be started within this method, not in the constructor. It can contain a blocking or non-blocking implementation.

Any extension of this method must first fork a call to the **`super.main()`** method.

### Example

*Example B-144*

```
task mii_mac_layer::main();
   super.main();
   ...
endtask: main
```

## vmm_xactor::new()

Creates an instance of the transactor base class.

### SystemVerilog

```
function new(string name,string instance,int stream_id = -
1,vmm_object parent);

With +define NO_VMM12

function new(string name,string instance,int
stream_id = -1);
```

### OpenVera

Not supported.

### Description

Creates an instance of the transactor base class, with the specified name, instance name, and optional stream identifier. The name and instance name are used to create the message service interface in the **`vmm_xactor::log`** property, and the specified stream identifier is used to initialize the **`vmm_xactor::stream_id`** property.

## vmm_xactor::notifications_e

Predefined notifications.

### SystemVerilog

```
typedef enum int {XACTOR_IDLE        = 999999,
                  XACTOR_BUSY        = 999998,
                  XACTOR_STARTED     = 999997,
                  XACTOR_STOPPED     = 999996,
                  XACTOR_RESET       = 999995,
                  XACTOR_STOPPING    = 999994,
                  XACTOR_IS_STOPPED  = 999993
                  } notifications_e;
```

### OpenVera

```
static int XACTOR_IDLE;
static int XACTOR_BUSY;
static int XACTOR_STARTED;
static int XACTOR_STOPPING;
static int XACTOR_STOPPED;
static int XACTOR_RESET;
```

### Description

Predefined notifications that are indicated, whenever the transactor changes state.

XACTOR_IDLE

ON or OFF notification that is indicated when the transactor is idle. Must be the complement of `XACTOR_BUSY`.

XACTOR_BUSY

ON or OFF notification that is indicated when the transactor is busy. Must be the complement of `XACTOR_IDLE`.

XACTOR_STARTED

`ONE_SHOT` notification indicating that the transactor is started.

XACTOR_STOPPING

ON or OFF notification indicating that a request is made for the transactor to stop.

XACTOR_STOPPED

`ONE_SHOT` notification indicating that all threads in the transactor are stopped.

XACTOR_RESET

`ONE_SHOT` notification indicating that the transactor is reset.

## Example

*Example B-145*

```
xactor.notify.wait_for(vmm_xactor::XACTOR_STARTED);
```

## vmm_xactor::notify

Notification service interface and pre-configure notifications.

### SystemVerilog

```
vmm_notify notify;
enum {XACTOR_IDLE;
      XACTOR_BUSY;
      XACTOR_STARTED;
      XACTOR_STOPPED;
      XACTOR_RESET;
     XACTOR_STOPPING;
     XACTOR_IS_STOPPED
     };
```

### OpenVera

Not supported.

### Description

Notification service interface and pre-configures notifications to indicate the state and state transitions of the transactor. The **`vmm_xactor::XACTOR_IDLE`** and **`vmm_xactor::XACTOR_BUSY`** notifications are **`vmm_notify::ON_OFF`**. All other events are **`vmm_notify::ONE_SHOT.`**

### Example

*Example B-146*

```
class consumer extends vmm_xactor;
   ...
   virtual task main();
      ...
      forever begin
```

```
        transaction tr;
        this.in_chan.peek(tr);
        tr.notify.indicate(vmm_data::STARTED);
        ...
        tr.notify.indicate(vmm_data::ENDED, ...);
        this.in_chan.get(tr);
      end
   endtask: main
endclass: consumer
```

## vmm_xactor::prepend_callback()

Prepends the specified callback façade instance with this instance of the transactor.

### SystemVerilog

```
virtual function void
           prepend_callback(vmm_xactor_callbacks cb);
```

### OpenVera

Not supported.

### Description

Callback methods are invoked in the order in which they were registered.

A warning is generated, if the same callback façade instance is registered more than once with the same transactor. A façade instance can be registered with more than one transactor. Callback façade instances can be unregistered and re-registered dynamically.

### Example

*Example B-147*

```
program test;
initial begin
   dut_env env = new;
   align_tx cb = new(...);
   env.build();
   foreach (env.mii[i]) begin
      env.mii[i].prepend_callback(cb);
   end
```

```
   env.run();
end
endprogram
```

## vmm_xactor::psdisplay()

Returns a human-readable description of the transactor.

### SystemVerilog

```
virtual function string psdisplay(string prefix = "")
```

### OpenVera

Not supported.

### Description

This method returns a human-readable description of the transactor. Each line is prefixed with the specified prefix.

### Example

*Example B-148*

```
class xactor extends vmm_xactor;
   ...
endclass

program prog;
   xactor xact = new;
   ...
   initial begin
      ...
      $display("Printing variables of Transactor\n %s \n",
         xact.psdisplay());
      ...
   end

endprogram
```

## vmm_xactor::register_vmm_sb_ds()

For more information on this method, refer to the *VMM Scoreboard User Guide*.

## vmm_xactor::reset_xactor()

Resets the state, and terminates the execution threads in this transactor instance.

### SystemVerilog

```
virtual function void
          reset_xactor(reset_e rst_typ = SOFT_RST);
```

### OpenVera

Not supported.

### Description

Resets the state, and terminates the execution threads in this transactor instance, according to the specified reset type (see Table B-1). The base class indicates the **`vmm_xactor::XACTOR_RESET`** and **`vmm_xactor::XACTOR_IDLE`** notifications, and resets the **`vmm_xactor::XACTOR_BUSY`** notification.

*Table B-1 Reset Types*

*Table B-2*

| Enumerated Value | Broadcasting Operation |
|---|---|
| vmm_xactor::SOFT_RST | Clears the content of all channels, resets all ON_OFF notifications, and terminates all execution threads. However, maintains the current configuration, notification service, and random number generation state information. The transactor must be restarted. This reset type must be implemented. |
| vmm_xactor::PROTOCOL_RST | Equivalent to a reset signaled through the physical interface. The information affected by this reset is user defined. |
| vmm_xactor::FIRM_RST | Like `SOFT_RST`, but resets all notification service interface and random-number-generation state information. This reset type must be implemented. |
| vmm_xactor::HARD_RST | Resets the transactor to the same state, found after construction. The registered callbacks are unregistered. |

To facilitate the implementation of this method, the actual values associated with these symbolic properties are of increasing magnitude (for example, **`vmm_xactor::FIRM_RST`** is greater than **`vmm_xactor::SOFT_RST`**). Not all reset types may be implemented by all transactors. Any extension of this method must call **`super.reset_xactor(rst_type)`** first to terminate the **`vmm_xactor::main()`** method, reset the notifications, and reset the main thread seed according to the specified reset type. Calling the **`super.reset_xactor()`** method with a reset type of **`vmm_xactor::PROTOCOL_RST`** is functionally equivalent to **`vmm_xactor::SOFT_RST`**.

## Example

*Example B-149*

```
function void
   mii_mac_layer::reset_xactor(reset_e typ = SOFT_RST);
   super.start_xactor(typ);
```

```
    ...
endfunction: reset_xactor
```

## vmm_xactor::restore_rng_state()

Restores the state of all random generators.

### SystemVerilog

```
virtual function void restore_rng_state();
```

### OpenVera

Not supported.

### Description

This method restores, from local properties, the state of all random generators associated with this transactor instance.

## vmm_xactor::stream_id

Identifier for the stream of transaction and data descriptors.

### SystemVerilog

```
int stream_id;
```

### OpenVera

Not supported.

### Description

The **`stream_id`** is a unique identifier for the stream of transaction and data descriptors, flowing through this transactor instance. It should be used to set the **`vmm_data::stream_id`** property of the descriptors, as they are received or randomized by this transactor.

### Example

*Example B-150*

```
class responder extends vmm_xactor;
   ...
   virtual task main();
      ...
      forever begin
         this.req_chan.get(tr);
         tr.stream_id = this.stream_id;
         tr.data_id   = response_id++;
         if (!tr.randomize()) ...
         ...
         this.resp_chan.sneak(tr);
      end
   endtask: main
endclass: responder
```

## vmm_xactor::save_rng_state()

Saves the state of all random generators.

### SystemVerilog

```
virtual function void save_rng_state();
```

### OpenVera

Not supported.

### Description

This method saves, in local properties, the state of all random generators associated with this transactor instance.

## vmm_xactor::start_xactor()

Starts the execution threads in this transactor instance.

### SystemVerilog

```
virtual function void start_xactor();
```

### OpenVera

Not supported.

### Description

Starts the execution threads in this transactor instance. The transactor can later be stopped. Any extension of this method must call the **`super.start_xactor()`** method. The base class indicates the **`vmm_xactor::XACTOR_STARTED`** and **`vmm_xactor::XACTOR_BUSY`** notifications, and resets the **`vmm_xactor::XACTOR_IDLE`** notification.

### Example

*Example B-151*

```
class tb_env extends vmm_env;
   ...
   virtual task start();
      super.start();
      ...
      this.mac.start_xactor();
      ...
   endtask: start
   ...
endclass: tb_env
```

## vmm_xactor::stop_xactor()

Stops the execution threads in this transactor instance.

### SystemVerilog

```
virtual function void stop_xactor();
```

### OpenVera

Not supported.

### Description

Stops the execution threads in this transactor instance. The transactor can later be restarted. Any extension of this method must call the **`super.stop_xactor() method`**. The transactor stops, when the **`vmm_xactor::wait_if_stopped()`** or **`vmm_xactor::wait_if_stopped_or_empty()`** method is called. It is a call to these methods to define the granularity of stopping a transactor.

## vmm_xactor::unregister_callback()

Unregisters the specified callback façade instance.

### SystemVerilog

```
virtual function void
           unregister_callback(vmm_xactor_callbacks cb);
```

### OpenVera

Not supported.

### Description

Unregisters the specified callback façade instance, for this transactor instance. A warning is generated, if the specified façade instance is not currently registered with the transactor. Callback façade instances can later be re-registered with the same or another transactor.

## vmm_xactor::unregister_vmm_sb_ds()

For more information on this method, refer to the *VMM Scoreboard User Guide*.

## vmm_xactor::‘vmm_callback()

Simplifies the syntax of invoking callback methods in a transactor.

### SystemVerilog

```
‘vmm_callback(callback_class_name, method(args));
```

### OpenVera

Not supported.

### Example

*Example B-152*

Instead of:

```
foreach (this.callbacks[i]) begin
   ahb_master_callbacks cb;
   if ($cast(cb, this.callbacks[i])) continue;
   cb.ptr_tr(this, tr, drop);
end
```

Use:

```
‘vmm_callback(ahb_master_callbacks, \
   ptr_tr(this, tr, drop));
```

## vmm_xactor::wait_if_stopped()

Suspends an execution thread.

### SystemVerilog

```
protected task wait_if_stopped(int unsigned n_threads = 1);
```

### OpenVera

```
protected task wait_if_stopped_t(integer n_threads = 1);
```

### Description

Blocks the thread execution, if the transactor is stopped through the `stop_xactor()` method. This method indicates the `vmm_xactor::XACTOR_STOPPED` and `vmm_xactor::XACTOR_IDLE` notifications, and resets the `vmm_xactor::XACTOR_BUSY` notification. The tasks will return, once the transactor is restarted using the `start_xactor()` method, and the specified input channel is not empty. These methods do not block, if the transactor is not stopped and the specified input channel is not empty.

Calls to this method and the "vmm_xactor::wait_if_stopped_or_empty()" methods define the granularity, by which the transactor can be stopped without violating the protocol. If a transaction can be suspended in the middle of its execution, then the `wait_if_stopped()` method should be called at every opportunity. If a transaction cannot be suspended, then the `wait_if_stopped_or_empty()` method should only be called after the current transaction is completed, before fetching the next transaction descriptor for the input channel.

If a transactor is implemented using more than one concurrently running thread that must be stopped, the total number of threads to be stopped must be specified in all invocations of this and the "vmm_xactor::wait_if_stopped_or_empty()" method.

## Example

*Example B-153*

```
protected virtual task main();
   super.main();
   forever begin
      transaction tr;
      this.wait_if_stopped_or_empty(this.in_chan);
      this.in_chan.activate(tr);
      ...
      this.wait_if_stopped();
      ...
   end
endtask: main
```

## vmm_xactor::wait_if_stopped_or_empty()

Suspends an execution thread or wait on a channel.

### SystemVerilog

```
protected task wait_if_stopped_or_empty(vmm_channel chan,
     int unsigned n_threads = 1);
```

### OpenVera

```
protected task wait_if_stopped_or_empty_t(rvm_channel chan,
     integer n_threads = 1);
```

### Description

Blocks the thread execution, if the transactor is stopped through the **`stop_xactor()`** method, or if the specified input channel is currently empty. This method indicates the **`vmm_xactor::XACTOR_STOPPED`** and **`vmm_xactor::XACTOR_IDLE`** notifications, and resets the **`vmm_xactor::XACTOR_BUSY`** notification. The tasks will return, once the transactor is restarted using the **`start_xactor()`** method, and the specified input channel is not empty. These methods do not block, if the transactor is not stopped and the specified input channel is not empty.

Calls to this method and the "vmm_xactor::wait_if_stopped()" methods define the granularity, by which the transactor can be stopped without violating the protocol.

If a transactor is implemented using more than one concurrently running thread that must be stopped, then the total number of threads to be stopped must be specified in all invocations of this and the "vmm_xactor::wait_if_stopped()" method.

## Example

*Example B-154*

```
protected virtual task main();
   super.main();
   fork
      forever begin
         transaction tr;
         this.wait_if_stopped_or_empty(this.in_chan, 2);
         this.in_chan.activate(tr);
         ...
         this.wait_if_stopped(2);
         ...
      end

      forever begin
         ...
         this.wait_if_stopped(2);
         ...
      end
   join_none
endtask: main
```

## vmm_xactor::xactor_status()

Displays the current status of the transactor instance.

### SystemVerilog

```
virtual function void xactor_status(string prefix = "");
```

### OpenVera

Not supported.

### Description

Displays the current status of the transactor instance in a human-readable format using the message service interface found in the **`vmm_log::log`** property, using the **`vmm_log::NOTE_TYP`** messages. Each line of the status information is prefixed with the specified prefix.

## ‘vmm_xactor_member_begin()

Starts the shorthand section.

### SystemVerilog

```
‘vmm_xactor_member_begin(class-name)
```

### OpenVera

Not supported.

### Description

Start the shorthand section, providing a default implementation for the `psdisplay()`, `start_xactor()`, `stop_xactor()`, and `reset_xactor()` methods.

The class-name specified must be the name of the `vmm_xactor` extension class that is being implemented.

The shorthand section can only contain shorthand macros, and must be terminated by the “‘vmm_xactor_member_end()” method.

### Example

*Example B-155*

```
class eth_mac extends vmm_xactor;
   ...
   ‘vmm_xactor_member_begin(eth_mac)
      ...
   ‘vmm_xactor_member_end(eth_mac)
   ...
endclass
```

## ‘vmm_xactor_member_end()

Terminates the shorthand section.

### SystemVerilog

```
‘vmm_xactor_member_end(class-name)
```

### OpenVera

Not supported.

### Description

Terminates the shorthand section, providing a default implementation for the `psdisplay()`, `start_xactor()`, `stop_xactor()`, and `reset_xactor()` methods.

The class-name specified must be the name of the `vmm_xactor` extension class that is being implemented.

The shorthand section must be started by the “‘vmm_xactor_member_begin()” method.

### Example

*Example B-156*

```
class eth_mac extends vmm_xactor;
   ...
   ‘vmm_xactor_member_begin(eth_mac)
      ...
   ‘vmm_xactor_member_end(eth_mac)
   ...
endclass
```

## ‘vmm_xactor_member_scalar*()

Shorthand implementation for a scalar data member.

### SystemVerilog

```
‘vmm_xactor_member_scalar(member-name,
                          vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_scalar_array(member-name,
                                vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_scalar_aa_scalar(member-name,
                                vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_scalar_aa_string(member-name,
                                vmm_xactor::do_what_e do_what)
```

### OpenVera

Not supported.

### Description

Adds the specified scalar-type, array of scalars, scalar-indexed associative array of scalars, or string-indexed associative array of scalars data member to the default implementation of the methods specified by the `’do_what’` argument.

A scalar is an integral type, such as bit, bit vector, and packed unions.

The shorthand implementation must be located in a section started by the `“‘vmm_xactor_member_begin()”` method.

## Example

*Example B-157*

```
class eth_frame_gen extends vmm_xactor;
   local integer fr_count;
   ...
   `vmm_xactor_member_begin(eth_frame_gen);
      `vmm_xactor_member_scalar (fr_count, DO_ALL)
      ...
   `vmm_xactor_member_end(eth_frame_gen)
   ...
endclass
```

## ‘vmm_xactor_member_string*()

Shorthand implementation for a string data member.

### SystemVerilog

```
‘vmm_xactor_member_string(member-name,
                          vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_string_array(member-name,
                                vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_string_aa_scalar(member-name,
                                vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_string_aa_string(member-name,
                                vmm_xactor::do_what_e do_what)
```

### OpenVera

Not supported.

### Description

Adds the specified string-type, array of strings, scalar-indexed associative array of strings, or string-indexed associative array of strings data member to the default implementation of the methods specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by the “‘vmm_xactor_member_begin()” method.

### Example

*Example B-158*

```
class eth_frame_gen extends vmm_xactor;
```

```
   local string fr_name;
   ...
   `vmm_xactor_member_begin(eth_frame_gen);
      `vmm_xactor_member_string (fr_name, DO_ALL)
      ...
   `vmm_xactor_member_end(eth_frame_gen)
   ...
endclass
```

## ‘vmm_xactor_member_enum*()

Shorthand implementation for an enumerated data member.

### SystemVerilog

```
‘vmm_xactor_member_enum(member-name,
                        vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_enum_array(member-name,
                              vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_enum_aa_scalar(member-name,
                              vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_enum_aa_string(member-name,
                              vmm_xactor::do_what_e do_what)
```

### OpenVera

Not supported.

### Description

Adds the specified enum-type, array of enums, scalar-indexed associative array of enums, or string-indexed associative array of enums data member to the default implementation of the methods specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by the “‘vmm_xactor_member_begin()” method.

### Example

*Example B-159*

```
class eth_frame_gen extends vmm_xactor;
```

```
   fr_type fr_type_var;
   ...
   `vmm_xactor_member_begin(eth_frame_gen);
      `vmm_xactor_member_enum (fr_type_var, DO_ALL)
      ...
   `vmm_xactor_member_end(eth_frame_gen)
   ...
endclass
```

## ‘vmm_xactor_member_vmm_data*()

Shorthand implementation for a vmm_data-based data member.

### SystemVerilog

```
‘vmm_xactor_member_vmm_data(member-name,
                            vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_vmm_data_array(member-name,
                            vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_vmm_data_aa_scalar(member-name,
                            vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_vmm_data_aa_string(member-name,
                            vmm_xactor::do_what_e do_what)
```

### OpenVera

Not supported.

### Description

Adds the specified vmm_data-type, array of `vmm_datas`, scalar-indexed associative array of `vmm_datas`, or string-indexed associative array of `vmm_datas` data member to the default implementation of the methods, specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by the `“‘vmm_xactor_member_begin()”` method.

## Example

*Example B-160*

```
class eth_frame extends vmm_data;
   ...
endclass

class eth_frame_gen extends vmm_xactor;
   eth_frame eth_frame_packet;
   ...
   `vmm_xactor_member_begin(eth_frame_gen);
     `vmm_xactor_member_vmm_data (eth_frame_packet, DO_ALL)
      ...
   `vmm_xactor_member_end(eth_frame_gen)
   ...
endclass
```

## ‘vmm_xactor_member_channel*()

Shorthand implementation for a channel data member.

### SystemVerilog

```
‘vmm_xactor_member_channel(member-name,
                           vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_channel_array(member-name,
                            vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_channel_aa_scalar(member-name,
                            vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_channel_aa_string(member-name,
                            vmm_xactor::do_what_e do_what)
```

### OpenVera

Not supported.

### Description

Adds the specified channel-type, array of channels, dynamic array of channels, scalar-indexed associative array of channels, or string-indexed associative array of channels data member to the default implementation of the methods specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by the `“‘vmm_xactor_member_begin()”` method.

## Example

*Example B-161*

```
class eth_frame_gen extends vmm_xactor;
   eth_frame_channel in_chan
   ...
   `vmm_xactor_member_begin(eth_frame_gen);
      `vmm_xactor_member_channel (in_chan, DO_ALL)
      ...
   `vmm_xactor_member_end(eth_frame_gen)
   ...
endclass
```

## ‘vmm_xactor_member_xactor*()

Shorthand implementation for a transactor data member.

### SystemVerilog

```
‘vmm_xactor_member_xactor(member-name,
                          vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_xactor_array(member-name,
                            vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_xactor_aa_scalar(member-name,
                            vmm_xactor::do_what_e do_what)

‘vmm_xactor_member_xactor_aa_string(member-name,
                            vmm_xactor::do_what_e do_what)
```

### OpenVera

Not supported.

### Description

Adds the specified transactor-type, array of transactors, dynamic array of transactors, scalar-indexed associative array of transactors, or string-indexed associative array of transactors data member to the default implementation of the methods, specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by the `“‘vmm_xactor_member_begin()”` method.

## Example

*Example B-162*

```
class custom_gen extends vmm_xactor;
   ...
endclass

class eth_frame_gen extends vmm_xactor;
  custom_gen custom_gen_inst;
   ...
   `vmm_xactor_member_begin(eth_frame_gen);
      `vmm_xactor_member_xactor (custom_gen_inst, DO_ALL)
      ...
   `vmm_xactor_member_end(eth_frame_gen)
   ...
endclass
```

## ‘vmm_xactor_member_user_defined()

User-defined shorthand implementation data member.

### SystemVerilog

```
‘vmm_xactor_member_user_defined(member-name)
```

### OpenVera

Not supported.

### Description

Adds the specified user-defined default implementation of the methods specified by the `’do_what’` argument.

The shorthand implementation must be located in a section started by the “‘vmm_xactor_member_begin()” method.

### Example

*Example B-163*

```
class eth_frame_gen extends vmm_xactor;
   integer fr_no;
   ...
   `vmm_xactor_member_begin(eth_frame_gen);
      `vmm_xactor_member_user_defined (fr_no, DO_ALL)
      ...
   `vmm_xactor_member_end(eth_frame_gen)
   function bit do_fr_no(input vmm_data::do_what_e do_what)
      do_fr_no = 1;  // Success, abort by returning 0
      case (do_what)
      endcase
   endfunction
endclass
```

## vmm_xactor_callbacks

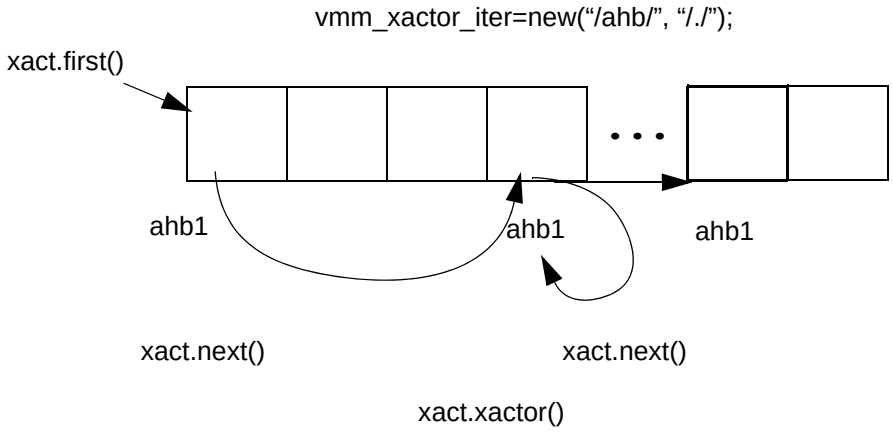
This class implements a virtual base class for callback containments. For more information, see the documentation for the “vmm_xactor::append_callback()” on page 418.

# vmm_xactor_iter

This class can iterate over all known vmm_xactor instances, based on the names and instance names, regardless of their location in the class hierarchy.

VMM adds this class to traverse list a registered transactors that match a regular expression. This feature is useful to register specific transactor callbacks, connect specific transactors to a scoreboard object, and re-allocate transactor, by killing its channels and reassigning some new ones.

```
class driver_typed #(type T = vmm_data) extends vmm_xactor;
  function new(string instance);
    super.new("driver", instance);
  endfunction
  virtual protected task main();
     vmm_channel chans[$];
     super.main();
     get_input_channels(chans);
     foreach (chans[i]) begin
       vmm_channel_typed #(T) chan;
       $cast(chan, chans[i]);
       start_drive(chan, i);
     end
  endtask
  virtual task start_drive(vmm_channel_typed #(T) chan);
    T tr;
    fork
     forever begin
       chan.get(tr);
       `vmm_note(log, tr.psdisplay("Executing.."));
       wait_if_stopped();
     end
   join_none
  endtask
endclass
```




VMM provides a method to access all transactors available in the environment using the **vmm_xactor_iter**. The VMM transactor iterator iterates over the transactors based on name or instances, using regexp. There is no need to know the hierarchical references to the **vmm_xactor** instance. The **vmm_xactor_iter** maintains a single queue of all transactors matching the specified regular expression.

The VMM transactor iterator can be used by either creating a new iterator object using **vmm_xactor_iter::new()** or by using the shorthand macro **`foreach_vmm_xactor()**. The two methods are explained below.

## Using the vmm_xactor_iter Class

```
vmm_xactor_iter iter = new("/./" , "/./");
```

Uses regexp style name and instance matching. "/./" returns all the **vmm_xactor** objects present in the environment.

The methods available with **vmm_xactor_iter** are:

- **`vmm_xactor_iter::first()`**

  Resets the iterator to the first transactor, that is to the start of the queue.

- **`vmm_xactor_iter::xactor()`**

  Returns a reference to the current transactor iterated on.

- **`vmm_xactor_iter::next()`**

  Moves the iterator to the next transactor.

The following below shows how to start all transactors extended from the `ahb_transactor` class. The `ahb_transactor` class is extended from the **`vmm_xactor`** class.

```
vmm_xactor_iter iter = new("/./", "/./");
   // Returns a list of all vmm_xactor objects
while( iter.xactor() != null) begin
   ahb_transactor ahb;
   if($cast(ahb, iter.xactor()) begin
      //get abh_transactor extended objects
      ahb.start_xactor();
   end
   iter.next();
end
```

## Using the Shorthand Macro `foreach_vmm_xactor()

The macro **`foreach_vmm_xactor`**`(ahb_transactor,"/./","/./")` requires three arguments. This call returns all objects of the `ahb_transactor` and its derived classes. For returning all **`vmm_xactor`** objects, use the **`vmm_xactor`** as the first argument. The second and third arguments are string name and instance, respectively.

The variable name, `xact`, of the type specified as the first argument is implicitly declared.

The following example achieves the same functionality as above, using the shorthand macro.

The macro must be used in the declarative portion of the code, or immediately followed by the **`begin`** keyword.

```
begin
   `foreach_vmm_xactor(ahb_transactor, "/./" , "/./")
   begin
      xact.start_xactor();
   end
end
```

## Summary


- vmm_xactor_iter::first() ..... page B-482
- vmm_xactor_iter::new() ..... page B-483
- vmm_xactor_iter::next() ..... page B-485
- vmm_xactor_iter::xactor() ..... page B-486
- `foreach_vmm_xactor() ..... page B-487

## vmm_xactor_iter::first()

Resets the iterator to the first transactor.

### SystemVerilog

```
function vmm_xactor first();
```

### OpenVera

Not supported.

### Description

Resets the iterator to the first transactor matching the name and instance name patterns specified, when the iterator was created using the `vmm_xactor_iter::new()` method and return a reference to it, if found.

Returns *NULL*, if no transactors match.

The order in which transactors are iterated on is unspecified.

### Example

*Example B-164*

```
int i = 0;
vmm_xactor_iter iter = new("/AHB/", "");
vmm_xactor xa;
for (xa = iter.first();  xa != null;  xa= iter.next())
   i++;
`vmm_note (log, $psprintf("No. of AHB transactors = %0d ",i))
```

## vmm_xactor_iter::new()

Creates a new transactor iterator.

### SystemVerilog

```
function void new(string name = "", string inst = "");
```

### OpenVera

Not supported.

### Description

Creates a new transactor iterato,r and initializes it using the specified name and instance name. If the specified name or instance name is enclosed between '/' characters, they are interpreted as regular expressions. Otherwise, they are interpreted as the full name or instance name to match.

The "vmm_xactor_iter::first()" is implicitly called. So, once created, the first transactor matching the specified name and instance name patterns is available, using the "vmm_xactor_iter::xactor()" method. The subsequent transactors can be iterated on, one at a time, using the "vmm_xactor_iter::next()" method.

### Example

*Example B-165*

```
vmm_xactor_iter iter = new("/AHB/");
while (iter.xactor() != null) begin
   ahb_master ahb;
   if ($cast(ahb, iter.xactor()) begin
```

```
        ...
    end
    iter.next();
end
```

## vmm_xactor_iter::next()

Moved the iterator to the next transactor.

### SystemVerilog

```
function vmm_xactor next();
```

### OpenVera

Not supported.

### Description

Moved the iterator to the next transactor, matching the name and instance name patterns specified, when the iterator was created using the `vmm_xactor_iter::new()` method and return a reference to it, if found.

Returns *NULL*, if no transactors match.

The order in which transactors are iterated on is unspecified.

### Example

*Example B-166*

```
int i = 0;
vmm_xactor_iter iter = new("/AHB/", "");
vmm_xactor xa;
for (xa = iter.first(); xa != null; xa= iter.next())
   i++;
`vmm_note (log, $psprintf("No. of AHB transactors = %0d ",i))
```

## vmm_xactor_iter::xactor()

Returns the current transactor iterated on.

### SystemVerilog

```
function vmm_xactor xactor();
```

### OpenVera

Not supported.

### Description

Returns a reference to a transactor, matching the name and instance name patterns specified ,when the iterator was created using the `vmm_xactor_iter::new()` method.

Returns *NULL*, if no transactors match.

### Example

*Example B-167*

```
vmm_xactor_iter iter = new("/AHB/");
while (iter.xactor() != null) begin
   ahb_master ahb;
   if ($cast(ahb, iter.xactor()) begin
      ...
   end
   iter.next();
end
```

## ‘foreach_vmm_xactor()

Shorthand transactor iterator macro.

### SystemVerilog

```
‘foreach_vmm_xactor(type, name, inst) begin
   xact...
end
```

### OpenVera

Not supported.

### Description

Shorthand macro to simplify the creation and operation of a transactor iterator instance, looking for transactors of a specific type, matching a specific name and instance name. The subsequent statement is executed for each transactor iterated on.

A variable named `"xact"` of the type specified as the first argument to the macro is implicitly declared, and iteratively set to each transactor of the specified type that matches the specified name and instance name.

The macro must be located immediately after a `"begin"` keyword.

### Example

*Example B-168 Iterating over all transactors of type "ahb_master"*

```
begin
   ‘foreach_vmm_xactor(ahb_master, "/./", "/./")
      begin
         xact.register_callback(...);
```

```
    end
end
```

# C

# Command Line Reference

This appendix provides detailed information about the command line references that compose the VMM Standard Library.

*Table C-1 Run-Time Switches*

| Option | Description |
|---|---|
| +vmm_break_on_phase | Specifies "+" separated list of phases on which to break |
| +vmm_break_on_timeline | Specifies "+" separated list of timelines on which to break |
| +vmm_channel_fill_thresh=<int> | GLOBAL option that sets the number of objects threshold in a channel. The default value is 10 |
| +vmm_channel_shared_log | All vmm_channel instances share the same log |
| +vmm_force_verbosity=[ERROR, WARNING, NORMAL, TRACE, DEBUG, VERBOSE] | Overrides the message verbosity level with the specified one |
| +vmm_ gen_rtl_config | Specifies Generation of VMM RTL Configuration |

*Table C-1 Run-Time Switches*

| Option | Description |
|---|---|
| +vmm_help | Lists all the VMM options specified through runtime command line and through the environment |
| +vmm_list_timeline | Lists the available timelines in simulation at the end of the pre-test timeline. This comes into effect when vmm_simulation::run_tests() is used to run the simulation |
| +vmm_list_phases | Lists the available phases in simulation at the end of the pre-test timeline. This comes into effect when vmm_simulation::run_tests() is used to run the simulation |
| +vmm_object_children_thresh=<int> | GLOBAL option that sets the number of child objects threshold. The default value is 100 |
| +vmm_object_root_thresh=<int> | GLOBAL option that sets the number of root objects threshold. The default value is 1000 |
| +vmm_ object_thresh_check | Global setting for checking object threshold in object hierarchy, channel, and scoreboard |
| +vmm_opts+enable_auto_start=0/1@<pattern> | Enables auto start of transactor with implicit phasing; Hierarchical control can be enabled through match patterns (enabled by default, 1) |
| +vmm_opts+enable_auto_stop=0/1@<pattern> | Enables auto stop of transactor with implicit phasing; ( disabled by default, 0) |
| +vmm_opts_file+filename | Passes a file to specify VMM runtime options |
| +vmm_opts+<option1>+<option2>+.... | Enables user to specify runtime options |
| +vmm_opts+pull_mode_on@<pattern> | Enables pull_mode for channels. Hierarchical control can be enabled through match patterns (push_mode by default) |
| +vmm_opts+stop_after_n_insts=[int]@<pattern> | Stops a specified atomic or scenario generator after running for the instances specified |
| +vmm_opts+stop_after_n_scenarios=[int]@<pattern> | Stops a scenario generator after running for the number of scenarios specified |

*Table C-1 Run-Time Switches*

| Option | Description |
|---|---|
| +vmm_log_debug | Enables debug of vmm_log |
| +vmm_log_default=[FATAL, ERROR, WARNING, NORMAL, TRACE, DEBUG, VERBOSE] | Sets the default message verbosity |
| +vmm_log_nowarn_at_200 | Suppresses warning message for more than 200 vmm_log instances creation |
| +vmm_log_nofatal_at_1000 | Suppresses fatal message for more than 1000 vmm_log instances creation |
| +vmm_rtl_config | Specifies VMM RTL Configuration option |
| +vmm_tr_verbosity=[NORMAL, TRACE, DEBUG, VERBOSE] | Enables the verbosity of transaction level debug |
| +vmm_test= <test> | Name of testcase(s) to run |
| +vmm_test_file=filename | Test cases specified in a file |
| +vmm_test_help | Lists available testcases |

Note:There should be only one `+vmm_opts+` in the runtime command line. You can provide any number of options with the same `vmm_opts`. For example,

```
./
simv+vmm_opts+enable_auto_start=0@env:drv0+stop_after_n
_insts=5+..
```

.

*Table C-2 Compile-Time Switches (+defines)*

| Option | Description |
|---|---|
| +define+VMM_11 | Enables support for only VMM 1.1 features |
| +define+VMM_IN_PACKAGE | Option that needs to be passed when the VMM Standard Library is to be embedded in a package inside an sv package |

*Table C-2 Compile-Time Switches (+defines)*

| Option | Description |
|---|---|
| +define+VMM_LOG_ANSI_COLOR | Colorizes messages based on severity (see vmm_log) |
| +define+VMM_LOG_FORMAT_FILE_LINE | Adds file name and line number to information provided by vmm_log |
| +define+VMM_NULL_LOG_MACROS | Compiles-out debug messages to gain every milligram of performance |
| +define+VMM_NO_NOTIFICATION | Compiles-out the notifications from vmm_data classes |
| +define+VMM_PARAM_CHANNEL | Turns-on Parameterized channel when not using VMM1.2 |
| +define+ VMM_POST_INCLUDE=filename | Includes a file, after vmm.sv is parsed to add customization macros (see chapter on VMM customization) |
| +define+VMM_PRE_INCLUDE=filename | Includes a file, before vmm.sv is parsed to add customization macros (see chapter on VMM customization) |
| +define+VMM_RAL_DATA_WIDTH | Defines the default RAL data width. The default value is 64 |
| +define+VMM_RW_ADDR_WIDTH | Defines the default RAL address width |
| +define+ VMM_SB_DS_IN_STDLIB | Enables the scoreboard integration methods with various standard library components. For example, channels, xactors, etc. |

# D

# Release Notes

## New Features in VMM User Guide

VMM 1.2 is default. However, you can enable VMM 1.1 only by using `+define+VMM_11` switch.

### New Base Classes

- Enhanced APIs in VMM Common Object
- Implicit phasing (`vmm_group`, `vmm_timeline`, `vmm_simulation`, `vmm_phase`, `vmm_phase_def`, `vmm_unit`)
- Tests with implicit phasing [`vmm_simulation::run_tests`]
- Multi-test concatenation

- Hierarchical options
- RTL configuration (`vmm_rtl_config`)
- VMM TLM base classes (`vmm_tlm*`)
- Parameterized channels and generators
- Parameterized datastream scoreboard
- vmm_connect
- vmm_notify_observer
- vmm_group
- Class factory (`` `vmm_class_factory ``)
- Transaction and environment debugging
- Simple match patterns